ATLAS's Accelerator Language Choice… *is C++*

Attila Krasznahorkay

The Setup

A	CS	14		HL-LHC Technology decision: CUDA or one of its less-proprietary competitors	3/31/2024				3/31/2024	Q1 2024	Will be concluded when developer documentation has been provided.
A	CS		14.1	Full parallelization pattern recommendation to collaboration	3/31/2024				3/31/2024	Q1 2024	
A	CS		14.2	Design patterns/tutorial on GPU migration	3/31/2024				3/31/2024	Q1 2024	

1.3.4 EF Tracking	Milestone	Choice of FPGA family	2024-03-21	2024-03-21
1.3.4 EF Tracking	Milestone	Choice of GPU language	2024-03-21	2024-03-21
1.3.4 EF Tracking	Milestone	Choice of host interface platform	2024-03-21	2024-03-21

- We have milestones defined for choosing a language to program GPUs with, in both the S&C R2R4 planning document and in TDAQ’s EF Tracking schedule
  - Today’s talk is meant to address both of these milestones, and put them both to rest (for the time being…)

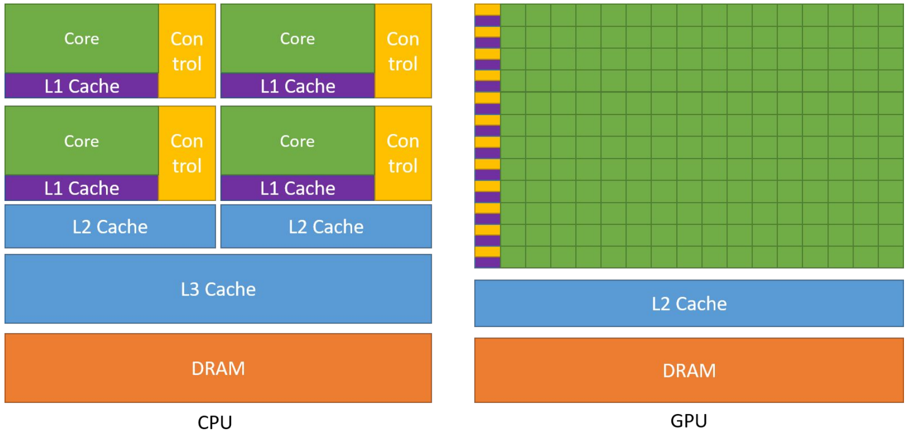
The Setup

- What this talk is meant to address is
  - How custom, hand-written algorithms, meant to run on GPUs, are to be written for the offline and trigger software
  - How people should organise their code
  - Just a little bit of how people are expected to build/test their code
- What this talk is **not** meant to address is
  - What sort of GPUs ATLAS would buy/use in the short and long terms
  - How we would deal with machine learning training and inference
  - How we would (efficiently) schedule the execution of hand-written GPU algorithms in our offline and trigger software

GPU Programming Basics

The Architecture

- GPUs have a lot more processing cores than CPUs do
  - And they can have orders of magnitudes more threads in flight than a CPU
- However all those cores are not nearly as independent as CPU cores are
  - Task based multithreading, like what we do in the offline/trigger code, does not fit to them
  - We must use a SIMT “approach” for our code
- Generally, we write functions (kernels) that would be executed on tens / hundreds of thousands of threads at the same time



Memory Management

- All “primary” languages provide low-level ways of (de-)allocating and copying memory
- Which APIs are fairly easy to write GPU SDK independent abstractions on top of
  - One such abstraction (vecmem) is available in Athena since December
- Generally, memory handling is not the biggest issue when choosing a GPU SDK / language

```
__host__ __device__ cudaError_t cudaMalloc ( void** devPtr , size_t size )
```

Allocate memory on the device.

Parameters

`devPtr`
- Pointer to allocated device memory

`size`
- Requested allocation size in bytes

```
void * malloc_device (size_t size, const queue &q, const property_list &propList, const detail::code_location &CodeLoc=detail::code_location::current())
void * aligned_alloc_device (size_t alignment, size_t size, const device &dev, const context &ctxt, const detail::code_location &CodeLoc=detail::code_location::current())
void * aligned_alloc_device (size_t alignment, size_t size, const device &dev, const context &ctxt, const property_list &propList, const detail::code_location &CodeLoc=detail::code_location::current())
void * aligned_alloc_device (size_t alignment, size_t size, const queue &q, const detail::code_location &CodeLoc=detail::code_location::current())
```

◆ jagged_vector_buffer() [3/3]

```
template<typename TYPE >
template<typename SIZE_TYPE , std::enable_if_t< std::is_integral< SIZE_TYPE >::value &&std::is_unsigned< SIZE_TYPE >::value, bool > >
vecmem::data::jagged_vector_buffer< TYPE >::jagged_vector_buffer ( const std::vector< SIZE_TYPE > & capacities,
                                                                   memory_resource &                 resource,
                                                                   memory_resource *                 host_access_resource = nullptr,
                                                                   buffer_type                       type = buffer_type::fixed_size
                                                                 )
```

Constructor from a vector of ("inner vector") sizes.

Parameters

capacities	Simple vector holding the capacities/sizes of the "inner vectors" for the jagged vector buffer.
resource	The device accessible memory resource, which may also be host accessible.
host_access_resource	An optional host accessible memory resource. Needed if `resource` is not host accessible.
type	The type (resizable or not) of the buffer

Writing / Launching Kernels

```
/// Kernel filling a jagged vector to its capacity
__global__ void fillTransformKernel(
    vecmem::data::jagged_vector_view<int> vec_data) {

    // Find the current index.
    const std::size_t i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if (i >= vec_data.size()) {
        return;
    }

    // Create a device vector on top of the view.
    vecmem::jagged_device_vector<int> vec(vec_data);

    // Fill the vectors to their capacity.
    while (vec[i].size() < vec[i].capacity()) {
        vec[i].push_back(1);
    }
}

void fillTransform(vecmem::data::jagged_vector_view<int> vec) {

    // Launch the kernel
    hipLaunchKernelGGL(fillTransformKernel, vec.size(), 1, 0, nullptr, vec);

    // Check whether it succeeded to run.
    VECMEM_HIP_ERROR_CHECK(hipGetLastError());
    VECMEM_HIP_ERROR_CHECK(hipDeviceSynchronize());
}
```

```
        // Now define the Alpaka Work division
        auto const deviceProperties = ::alpaka::getAccDevProps<Acc>(devAcc);
        auto const maxThreadsPerBlock = deviceProperties.m_blockThreadExtentMax[0];
        auto const threadsPerBlock = maxThreadsPerBlock;
        auto const blocksPerGrid =
            (sp_size + threadsPerBlock - 1) / threadsPerBlock;
        auto const elementsPerThread = 1u;
        auto workDiv = WorkDiv{blocksPerGrid, threadsPerBlock, elementsPerThread};
        auto bufAcc = ::alpaka::allocBuf<float, uint32_t>(devAcc, sp_size);

        ::alpaka::exec<Acc>(queue, workDiv, CountGridCapacityKernel{}, m_config,
                            &m_axes.first, &m_axes.second, spacepoints_view,
                            &grid_capacities_view);
        ::alpaka::wait(queue);
```

- In all cases we write "some function"
    - Depending on the language this may need to be a standalone function, or could be even something like a functor or lambda
    - The compiler needs to recognize it as a "kernel" function, to generate the appropriate binaries for it
- We tell "some runtime API" to launch this function, with a set of arguments, on a selected number of GPU threads

Device Code

- Is mostly pretty standard C(++)
  - “Kernel” functions can call as many “device” functions as they wish, just like in regular C++
- With (mostly) the following extensions:
  - Atomic operations on memory shared by all / some of the threads
  - Cooperative usage of memory dedicated to a block of threads
  - Synchronization points between the threads
- Other, language specific features also exist, but were not needed in the tracking R&D so far
  - The calo clusterization R&D code uses one CUDA specific feature right now that we’ll need to see about…

```
namespace traccc::device {

TRACCC_HOST_DEVICE
inline void find_doublets(
    const std::size_t globalIndex, const seedfinder_config& config,
    const sp_grid_const_view& sp_view,
    const doublet_counter_collection_types::const_view& dc_view,
    device_doublet_collection_types::view mb_doublets_view,
    device_doublet_collection_types::view mt_doublets_view) {

    // Check if anything needs to be done.
    const doublet_counter_collection_types::const_device doublet_counts(
        dc_view);
    if (globalIndex >= doublet_counts.size()) {
        return;
    }

    // Get the middle spacepoint that we need to be looking at.
    const doublet_counter middle_sp_counter

    // Set up the device containers.
    const const_sp_grid_device sp_grid(sp_v
    device_doublet_collection_types::device
    device_doublet_collection_types::device

    // Get the spacepoint that we're evalua
    // as the "middle" spacepoint.
    const internal_spacepoint<spacepoint> m
        sp_grid.bin(middle_sp_counter.m_spM
            .at(middle_sp_counter.m_spM.sp_

    // Find the reference (start) index of
    // where the doublets are recorded.
    const unsigned int mid_bot_start_idx =
    const unsigned int mid_top_start_idx =
```

```
static __global__
void signalToNoiseKernel(Helpers::CUDA_kernel_object<CellStateArr> cell_state_arr,
                         Helpers::CUDA_kernel_object<ClusterInfoArr> clusters_arr,
                         Helpers::CUDA_kernel_object<TopoAutomatonGrowingTemporaries> temporaries,
                         const Helpers::CUDA_kernel_object<CellInfoArr> cell_info_arr,
                         const Helpers::CUDA_kernel_object<CellNoiseArr> noise_arr,
                         const Helpers::CUDA_kernel_object<GeometryArr> geometry,
                         const Helpers::CUDA_kernel_object<TopoAutomatonOptions> opts)
{
  const int index = blockIdx.x * blockDim.x + threadIdx.x;

  const int grid_size = gridDim.x * blockDim.x;

  for (int cell = index; cell < NCaloCells; cell += grid_size)
    {
      const int cell_sampling = geometry->sampling(cell);
      const float cellEnergy = cell_info_arr->energy[cell];

      if (!cell_info_arr->is_valid(cell) || !opts->uses_calorimeter_by_sampling(cell_sampling))
        {
          cell_state_arr->clusterTag[cell] = TACTag::make_invalid_tag();
          temporaries->secondary_array[cell] = TACTag::make_invalid_tag();
          continue;
        }

      float sigNoiseRatio = 0.00001f;
      //It's what's done in the CPU implementation...
      if (!cell_info_arr->is_bad(cell, opts->treat_L1_predicted_as_good))
        {
          const int gain = cell_info_arr->gain[cell];

          float cellNoise = 0.f;

          if (opts->use_two_gaussian && geometry->is_tile(cell))
            {
              cellNoise = noise_arr->get_double_gaussian_noise(cell, gain, cellEnergy);
            }
          else
```

GPU R&D Takeaways

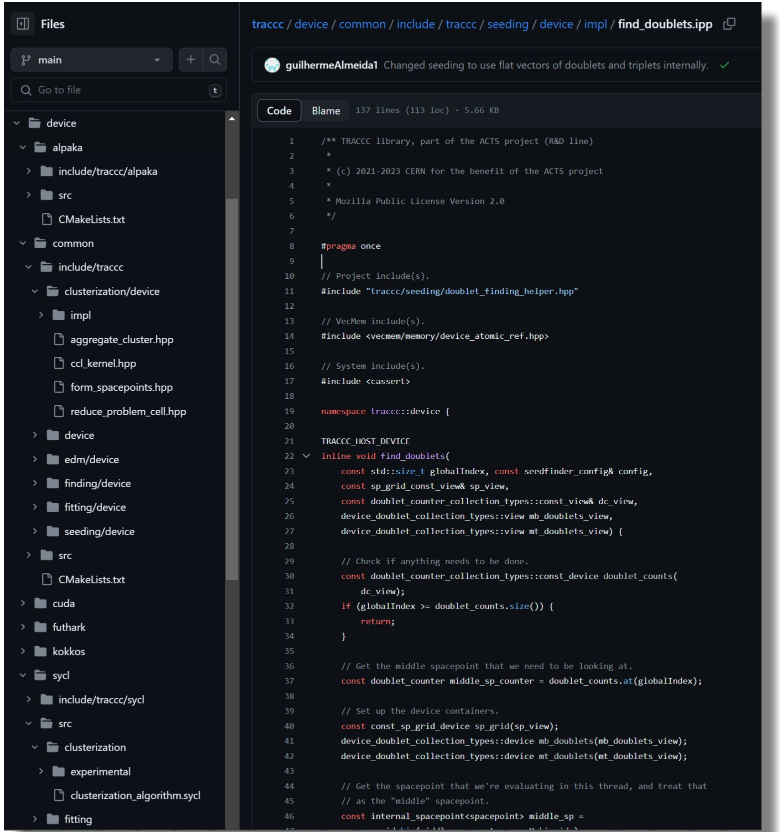
Considered / Tested Languages

	CUDA	HIP	SYCL	Kokkos	Alpaka	std::par
NVIDIA	✅	✅	✅	✅	✅	✅
AMD	❌	✅	✅	✅	✅	✅
Intel	❌	❌	✅	✅	✅	✅

- The “support matrix” is a bit misleading
  - To produce NVIDIA, AMD and Intel binaries, you must have CUDA, HIP and oneAPI available respectively. Something like Alpaka, or even oneAPI, needs CUDA in the background for producing NVIDIA binaries!
- I.e. the “abstraction layers” don’t help a lot with “platform support” or licensing 🙁

The `traccc` Lesson(s)

- In the GPU Tracking R&D (`traccc`) we've set up a very large amount of code sharing between the different GPU languages
  - Just by carefully thinking it over how we should organise the source code
- The amount of language specific code is not negligible, but can be written/translated **very** automatically
  - Didn't try it myself with `traccc`, but HIPIFY, DPCT, SYCLomatic, etc. could even do >90% of this automated work when/if needed.



The traccc Lesson(s)

Generic C++

```cpp
// Project include(s).
#include "traccc/definitions/qualifiers.hpp"
#include "traccc/device/fill_prefix_sum.hpp"
#include "traccc/edm/device/doublet_counter.hpp"
#include "traccc/seeding/detail/seeding_config.hpp"
#include "traccc/seeding/detail/spacepoint_grid.hpp"

// System include(s).
#include <cstddef>

namespace traccc::device {

/// Function used for calculating the number of spacepoint doublets
///
/// The count is necessary for allocating the appropriate amount of memory
/// for storing the information of the candidates in a next step.
///
/// @param[in] globalIndex   The index of the current thread
/// @param[in] config        Seedfinder configuration
/// @param[in] sp_view       The spacepoint grid to count doublets on
/// @param[in] sp_ps_view    Prefix sum for iterating over the spacepoint grid
/// @param[out] doublet_view Collection storing the number of doublets for each
/// spacepoint
/// @param[out] nMidBot      Total number of middle-bottom doublets
/// @param[out] nMidTop      Total number of middle-top doublets
///
TRACCC_HOST_DEVICE
inline void count_doublets(
    std::size_t globalIndex, const seedfinder_config& config,
    const sp_grid_const_view& sp_view,
    const vecmem::data::vector_view<const prefix_sum_element_t>& sp_ps_view,
    doublet_counter_collection_types::view doublet_view, unsigned int& nMidBot,
    unsigned int& nMidTop);

}  // namespace traccc::device

// Include the implementation.
#include "traccc/seeding/device/impl/count_doublets.ipp"
```

CUDA

```cpp
namespace traccc::cuda {
namespace kernels {

/// CUDA kernel for running @c traccc::device::count_doublets
__global__ void count_doublets(
    seedfinder_config config, sp_grid_const_view sp_grid,
    vecmem::data::vector_view<const device::prefix_sum_element_t> sp_prefix_sum,
    device::doublet_counter_collection_types::view doublet_counter,
    unsigned int& nMidBot, unsigned int& nMidTop) {

    device::count_doublets(threadIdx.x + blockIdx.x * blockDim.x, config,
                           sp_gr
                           nMidT
}
```

```cpp
        // Count the number of doublets that we need to produce.
        kernels::count_doublets<<<nDoubletCountBlocks, nDoubletCountThreads, 0,
                                  stream>>>(
            m_seedfinder_config, g2_view, sp_grid_prefix_sum_buff,
            doublet_counter_buffer, (*globalCounter_device).m_nMidBot,
            (*globalCounter_device).m_nMidTop);
        CUDA_ERROR_CHECK(cudaGetLastError());
```

SYCL

```cpp
        details::get_queue(m_queue)
            .submit([&](::sycl::handler& h) {
                h.parallel_for<kernels::count_doublets>(
                    doubletCountRange,
                    [config = m_seedfinder_config, g2_view, sp_grid_prefix_sum_view,
                     doublet_counter_view,
                     aux_globalCounter](::sycl::nd_item<1> item) {
                        device::count_doublets(item.get_global_linear_id(), config,
                                               g2_view, sp_grid_prefix_sum_view,
                                               doublet_counter_view,
                                               (*aux_globalCounter).m_nMidBot,
                                               (*aux_globalCounter).m_nMidTop);
                    });
            })
            .wait_and_throw();
```

Conclusions

The Choice of No Choice

- Instead of buying into a specific SDK, we have to structure all our new GPU code such as to make it easy/trivial to use different SDKs with the same “core” code
  - Performance penalties for using a “non-native” SDK are pretty minimal at the moment, but we will continue monitoring this
- Depending on how licensing and technical developments go, we may very well come out with a recommended SDK in the end
  - But even at that point, code will be structures so that it would still be easy to use from other SDKs as well at a later date
- For now, CUDA will be the easiest to use with Athena nightlies, inside the CERN firewall
  - oneAPI can be used already today from CVMFS, with a bit of manual environment setup
    - Even with GCC 13! Today!
  - HIP can not be installed on CVMFS just yet, but hopefully soon…

Summary

- If you're just now starting out, have a look at:
  - Control/AthenaExamples/AthExCUDA
  - Control/AthenaExamples/AthExSYCL
  - Will add slightly more elaborate examples, with code sharing between CUDA and SYCL, in not too long
- If you have a working setup already, just continue using it
  - Though if you're not in contact with people from HCAF, please get in touch with us! To make sure that your code would be future-proof.

Backup

Code Snippets

```cpp
namespace traccc::device {

TRACCC_HOST_DEVICE
inline void form_spacepoints(
    const std::size_t globalIndex,
    measurement_collection_types::const_view measurements_view,
    cell_module_collection_types::const_view modules_view,
    const unsigned int measurement_count,
    spacepoint_collection_types::view spacepoints_view) {

    // Get device copy of input parameters
    const measurement_collection_types::const_device measurements_device(
        measurements_view);

    // Check if anything needs to be done
    if (globalIndex >= measurement_count) {
        return;
    }

    // Get device copy of input parameters
    const cell_module_collection_types::const_device modules_device(
        modules_view);

    spacepoint_collection_types::device spacepoints_device(spacepoints_view);

    // Get the measurement for this index
    const measurement& meas = measurements_device.at(globalIndex);
    // Get the current cell module
    const cell_module& mod = modules_device.at(meas.module_link);
    // Form a spacepoint based on this measurement
    point3 local_3d = {meas.local[0], meas.local[1], 0.};
    point3 global = mod.placement.point_to_global(local_3d);

    // Fill the result object with this spacepoint
    spacepoints_device[globalIndex] = {global, meas};
}

}  // namespace traccc::device
```

```cpp
namespace traccc::device {

TRACCC_HOST_DEVICE
inline void count_doublets(
    const std::size_t globalIndex, const seedfinder_config& config,
    const sp_grid_const_view& sp_view,
    const vecmem::data::vector_view<const prefix_sum_element_t>& sp_ps_view,
    doublet_counter_collection_types::view doublet_view, unsigned int& nMidBot,
    unsigned int& nMidTop) {

    // Check if anything needs to be done.
    vecmem::device_vector<const prefix_sum_element_t> sp_prefix_sum(sp_ps_view);
    if (globalIndex >= sp_prefix_sum.size()) {
        return;
    }

    // Get the middle spacepoint that we need to be looking at.
    const prefix_sum_element_t middle_sp_idx = sp_prefix_sum[globalIndex];

    // Set up the device containers.
    const const_sp_grid_device sp_grid(sp_view);
    doublet_counter_collection_types::device doublet_counter(doublet_view);

    // Get the spacepoint that we're evaluating in this thread, and treat that
    // as the "middle" spacepoint.
    const internal_spacepoint<spacepoint> middle_sp =
        sp_grid.bin(middle_sp_idx.first).at(middle_sp_idx.second);

    // The the IDs of the neighbouring bins along the phi and Z axes of the
    // grid.
    const detray::dindex_range phi_bins =
        sp_grid.axis_p0().range(middle_sp.phi(), config.neighbor_scope);
    const detray::dindex_range z_bins =
        sp_grid.axis_p1().range(middle_sp.z(), config.neighbor_scope);
    assert(z_bins[0] <= z_bins[1]);

    // The number of middle-bottom candidates found for this thread's middle
    // spacepoint.
    unsigned int n_mb_cand = 0;
    // The number of middle-top candidates found for this thread's middle
    // spacepoint.
    unsigned int n_mt_cand = 0;
```

```cpp
template <typename barrier_t>
TRACCC_DEVICE inline void ccl_kernel(
    const index_t threadId, const index_t blckDim, const unsigned int blockId,
    const cell_collection_types::const_view cells_view,
    const cell_module_collection_types::const_view modules_view,
    const index_t max_cells_per_partition,
    const index_t target_cells_per_partition, unsigned int& partition_start,
    unsigned int& partition_end, unsigned int& outi, index_t* f, index_t* gf,
    barrier_t& barrier, measurement_collection_types::view measurements_view,
    unsigned int& measurement_count,
    vecmem::data::vector_view<unsigned int> cell_links) {

    // Get device copy of input parameters
    const cell_collection_types::const_device cells_device(cells_view);
    const cell_module_collection_types::const_device modules_device(
        modules_view);
    measurement_collection_types::device measurements_device(measurements_view);

    const unsigned int num_cells = cells_device.size();

    /*
     * First, we determine the exact range of cells that is to be examined
     * by this block of threads. We start from an initial range determined
     * by the block index multiplied by the target number of cells per
     * block. We then shift both the start and the end of the block forward
     * (to a later point in the array); start and end may be moved different
     * amounts.
     */
    if (threadId == 0) {
        unsigned int start = blockId * target_cells_per_partition;
        assert(start < num_cells);
        unsigned int end =
            std::min(num_cells, start + target_cells_per_partition);
        outi = 0;

        /*
         * Next, shift the starting point to a position further in the
         * array; the purpose of this is to ensure that we are not operating
         * on any cells that have been claimed by the previous block (if
         * any).
         */
        while (start != 0 &&
               cells_device[start - 1].module_link ==
                   cells_device[start].module_link &&
               cells_device[start].channel1 <=
                   cells_device[start - 1].channel1 + 1) {
            ++start;
```

Code Snippets

```cpp
/// CUDA kernel for running @c traccc::device::ccl_kernel
__global__ void ccl_kernel(
    const cell_collection_types::const_view cells_view,
    const cell_module_collection_types::const_view modules_view,
    const index_t max_cells_per_partition,
    const index_t target_cells_per_partition,
    measurement_collection_types::view measurements_view,
    unsigned int& measurement_count,
    vecmem::data::vector_view<unsigned int> cell_links) {
    __shared__ unsigned int partition_start, partition_end;
    __shared__ unsigned int outi;
    extern __shared__ index_t shared_v[];
    index_t* f = &shared_v[0];
    index_t* f_next = &shared_v[max_cells_per_partition];
    traccc::cuda::barrier barry_r;

    device::ccl_kernel(threadIdx.x, blockDim.x, blockIdx.x, cells_view,
                       modules_view, max_cells_per_partition,
                       target_cells_per_partition, partition_start,
                       partition_end, outi, f, f_next, barry_r,
                       measurements_view, measurement_count, cell_links);
}
```

```cpp
        // Launch ccl kernel. Each thread will handle a single cell.
        kernels::
            ccl_kernel<<<num_partitions, threads_per_partition,
                         2 * max_cells_per_partition * sizeof(index_t), stream>>>(
                cells, modules, max_cells_per_partition,
                m_target_cells_per_partition, measurements_buffer,
                *num_measurements_device, cell_links);
```

```cpp
        // Run ccl kernel
        details::get_queue(m_queue)
            .submit([&](::sycl::handler& h) {
                vecmem::sycl::local_accessor<unsigned int> shared_uint(3, h);
                vecmem::sycl::local_accessor<index_t> shared_idx(
                    2 * max_cells_per_partition, h);

                h.parallel_for<kernels::ccl_kernel>(
                    cclKernelRange, [=](::sycl::nd_item<1> item) {
                        index_t* f = &shared_idx[0];
                        index_t* f_next = &shared_idx[max_cells_per_partition];
                        unsigned int& partition_start = shared_uint[0];
                        unsigned int& partition_end = shared_uint[1];
                        unsigned int& outi = shared_uint[2];
                        traccc::sycl::barrier barry_r(item);

                        device::ccl_kernel(
                            item.get_local_linear_id(), item.get_local_range(0),
                            item.get_group_linear_id(), cells, modules,
                            max_cells_per_partition, target_cells_per_partition,
                            partition_start, partition_end, outi, f, f_next,
                            barry_r, measurements_view,
                            *aux_num_measurements_device, cell_links_view);
                    });
            })
            .wait_and_throw();
```

CERN

http://home.cern