Heterogeneous Memory Management with VecMem

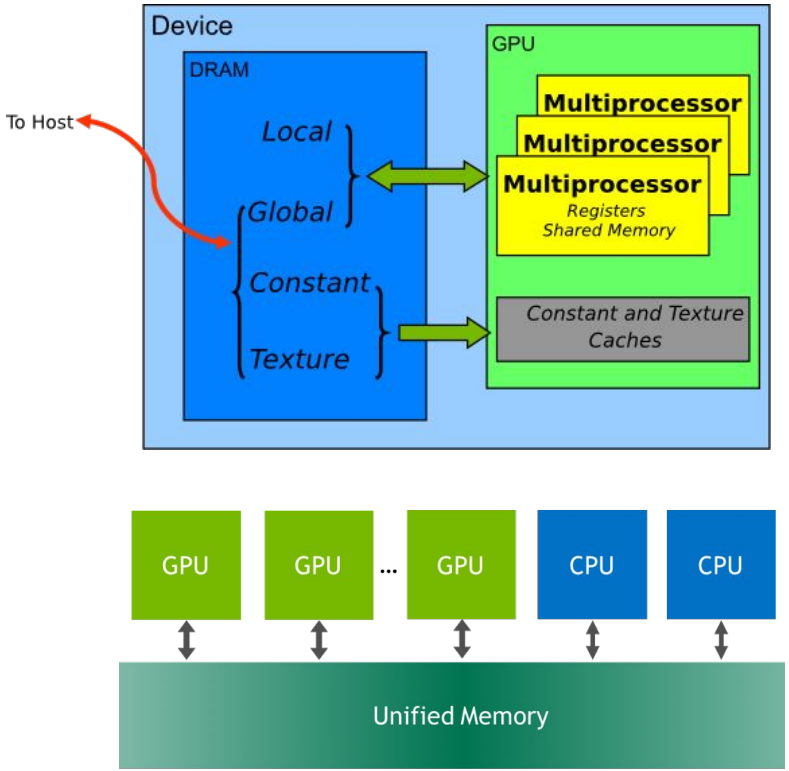
Attila Krasznahorkay, Stephen Swatman, Paul Gessinger

Overview

- Memory management on accelerators
- C++ memory resources
- Memory management with memory resources
  - Organising the memory handling in host code
  - Organising the memory handling in device code
- Uses of VecMem in algorithmic code

Accelerator Memory Management

- Modern CPUs have a very complicated memory management system
  - Which we can in most cases avoid knowing about
- GPUs have a complicated system of their own
  - However this we can not avoid knowing more about to use GPUs efficiently ☹
  - Most importantly, caching is much less automated than on modern CPUs
- In some cases however you can get away with not knowing everything
  - For a performance penalty...



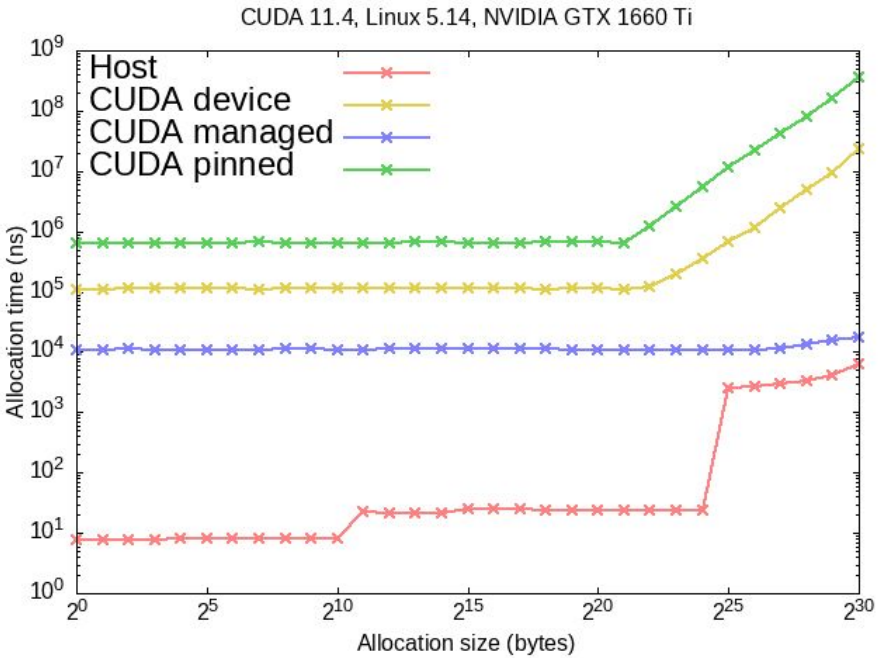
Memory Management APIs

```
const std::size_t arraySize = …;
float *hostArray = nullptr,
      *deviceArray = nullptr;
hostArray =
   (float*) malloc( arraySize * sizeof(float) );
CUDA_CHECK(
   cudaMalloc( &deviceArray,
               arraySize * sizeof(float) ) );
…
CUDA_CHECK(
   cudaMemcpy( deviceArray, hostArray,
               arraySize * sizeof(float),
               cudaMemcpyHostToDevice ) );
doSomething<<<…>>>(…, deviceArray, …);
…
free( hostArray );
CUDA_CHECK( cudaFree( deviceArray ) );
```

- All “heterogeneous platforms” provide memory management functions through a C API
  - Even in oneAPI/SYCL this is becoming the dominant way for managing memory
- This is very understandable, as it allows for interfacing with practically any other language
- However it is very different from how you are supposed to manage memory in modern C++
  - You shouldn’t even be using new/delete, let alone std::malloc(…)/std::free(…)

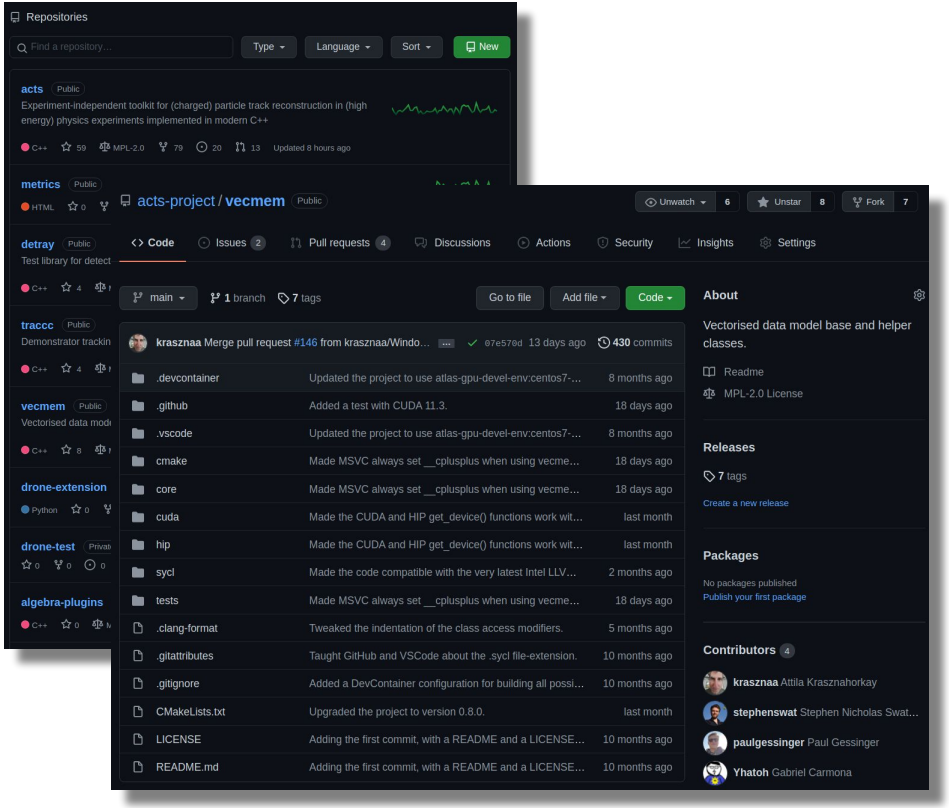
Device Memory Allocation Performance

- Using the CUDA/HIP/SYCL “C API” is not only inconvenient, but it is also significantly slower than host memory allocation
- This is one of the reasons why we generally can’t handle device memory with the same flexibility as we can host memory
  - Naive `push_back(...)`-s into a vector would be a performance killer

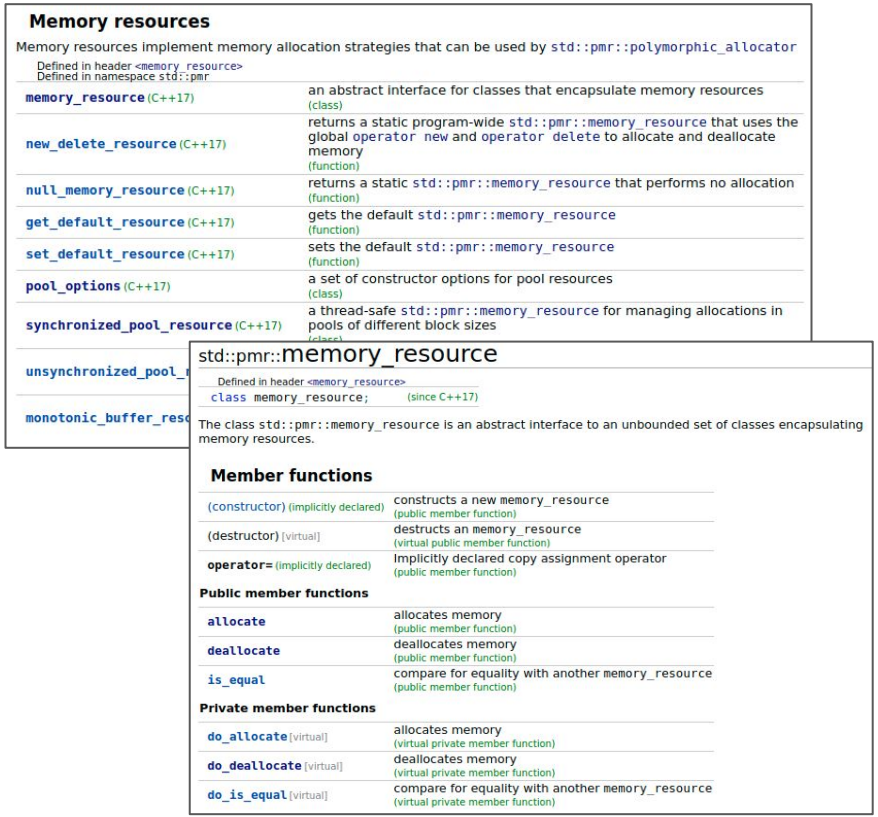


The VecMem Project

- In ATLAS we are currently working on a “full chain demonstrator” for running charged particle tracking on GPUs
  - The “top level” project being acts-project/traccc
  - As the development started last year, it quickly became clear that we needed to break the issue down into separate domains
    - We ended up separating the code for memory management (VecMem), algebra abstraction and detector geometry description out of the “top level” repository



C++17 Memory Resources



Memory resources

Memory resources implement memory allocation strategies that can be used by `std::pmr::polymorphic_allocator`

Defined in header `<memory_resource>`
Defined in namespace `std::pmr`

`memory_resource` (C++17)	an abstract interface for classes that encapsulate memory resources (class)
`new_delete_resource` (C++17)	returns a static program-wide `std::pmr::memory_resource` that uses the global `operator new` and `operator delete` to allocate and deallocate memory (function)
`null_memory_resource` (C++17)	returns a static `std::pmr::memory_resource` that performs no allocation (function)
`get_default_resource` (C++17)	gets the default `std::pmr::memory_resource` (function)
`set_default_resource` (C++17)	sets the default `std::pmr::memory_resource` (function)
`pool_options` (C++17)	a set of constructor options for pool resources (class)
`synchronized_pool_resource` (C++17)	a thread-safe `std::pmr::memory_resource` for managing allocations in pools of different block sizes (class)
`unsynchronized_pool_r`	
`monotonic_buffer_reso`	

std::pmr::memory_resource

Defined in header `<memory_resource>`

```
class memory_resource;        (since C++17)
```

The class `std::pmr::memory_resource` is an abstract interface to an unbounded set of classes encapsulating memory resources.

Member functions

(constructor) (implicitly declared)	constructs a new `memory_resource` (public member function)
(destructor) [virtual]	destructs an `memory_resource` (virtual public member function)
`operator=` (implicitly declared)	Implicitly declared copy assignment operator (public member function)

Public member functions

`allocate`	allocates memory (public member function)
`deallocate`	deallocates memory (public member function)
`is_equal`	compare for equality with another `memory_resource` (public member function)

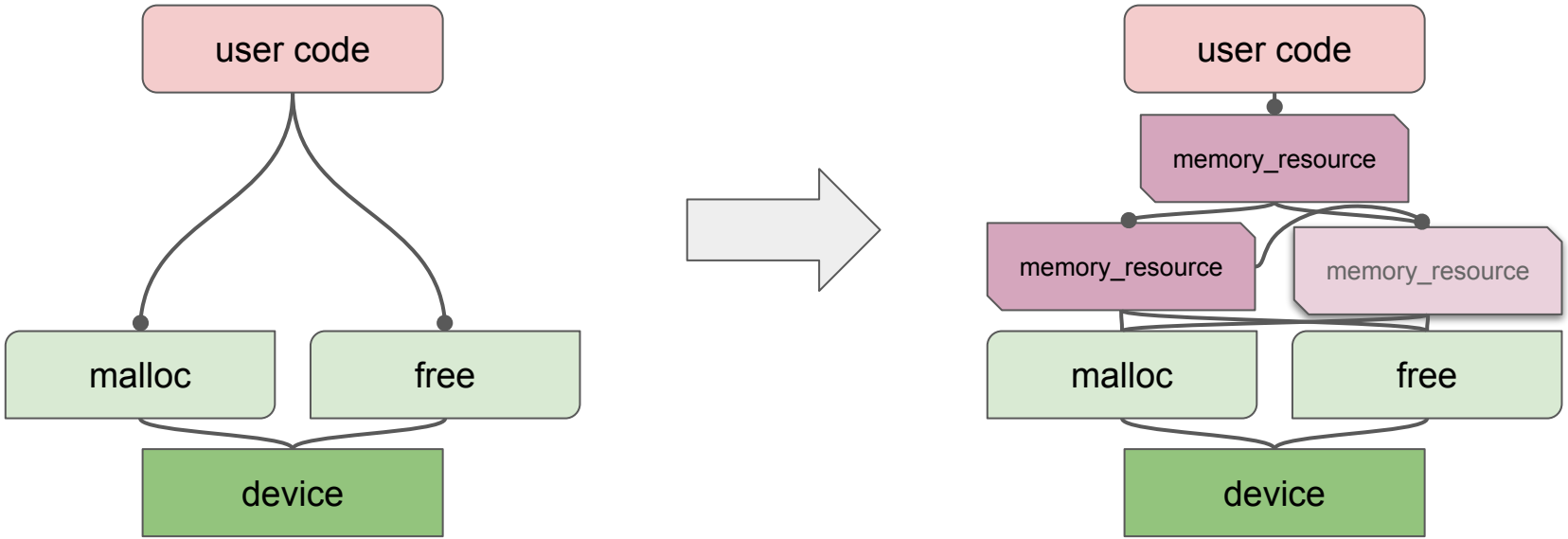
Private member functions

`do_allocate` [virtual]	allocates memory (virtual private member function)
`do_deallocate` [virtual]	deallocates memory (virtual private member function)
`do_is_equal` [virtual]	compare for equality with another `memory_resource` (virtual private member function)

- C++17 introduced a set of new classes/interfaces in the std::pmr namespace for generalised memory management
  - With a design having its roots in Thrust
- Unfortunately it is taking some time to be implemented in lib(std)c++
  - libstdc++ implemented full support in version 9.1
  - libc++ still did not fully implement it
    - We are working around that in VecMem

Allocation Through Composition

- STL classes can perform all their memory allocations through the abstract std::pmr::memory_resource interface
  - But implementations of that interface do not need to do all the work by themselves. They can also delegate work to other resources through the same abstract interface.



Upstream / Downstream Memory Resources

- One can categorise memory resources into two groups. We use the following nomenclature for now.
- Upstream resources, which talk “directly” to some device/memory
  - We have implementations for all (“main”) types of memory that CUDA/HIP/SYCL have to offer
- Downstream resources, which only talk to other resources
  - We have multiple caching resources, some that can perform “logic operations”, and some with “side effects”
- Technically “hybrid” resources could also be made, but they should not be needed. Everything can be expressed with just the two types that we have at the moment.

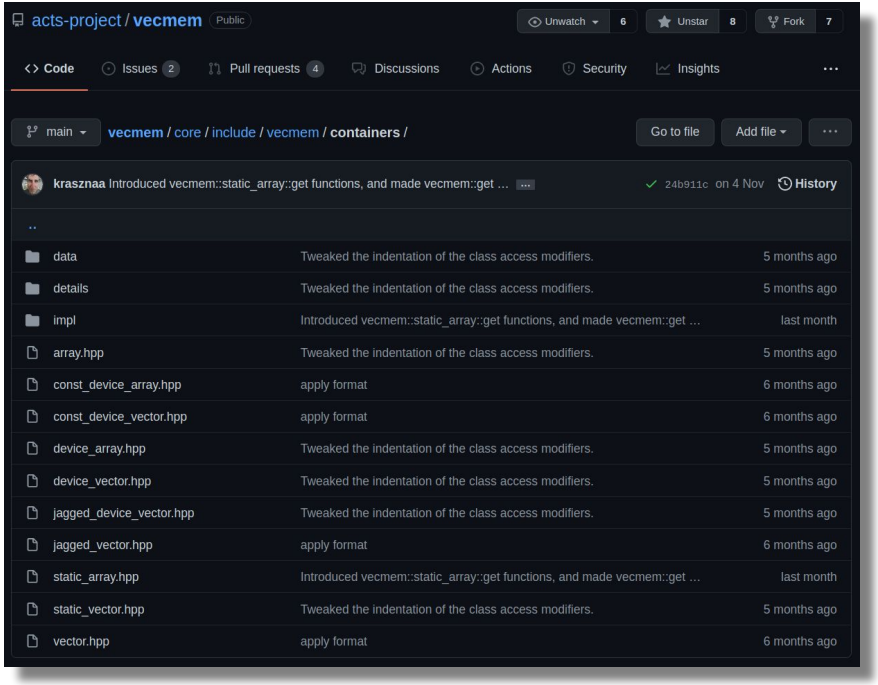
Memory Resource Usage Examples

```
vecmem::cuda::managed_memory_resource cuda_mr;
vecmem::arena_memory_resource arena_mr(cuda_mr);

vecmem::vector<float> floats(&arena_mr);
floats.resize(100, 5.f);

superKernel<<<100, 1>>>(vecmem::get_data(floats));
```

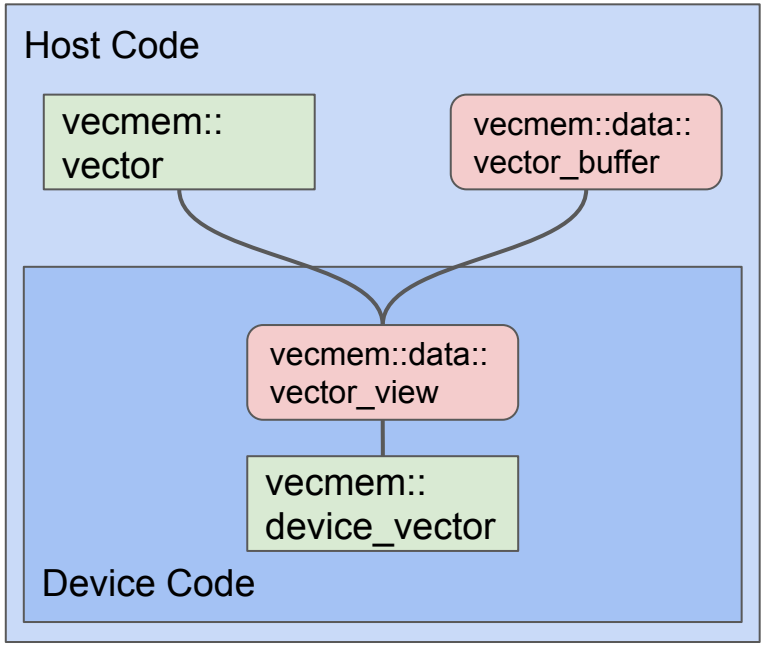
```
vecmem::host_memory_resource host_mr;
vecmem::cuda::device_memory_resource device_mr;

vecmem::vector<int> ints(&host_mr);
ints.resize(100, 5);

vecmem::cuda::copy copy;
vecmem::data::vector_buffer<int> device_ints =
   copy.to(vecmem::get_data(ints), device_mr);

hyperKernel<<<100, 1>>>(
   vecmem::get_data(device_ints));
```

Device Containers

- Managing device memory in host code using an STL syntax is just half of the story
- We also want to be able to interact with this memory in device code, with the same STL syntax
  - Since the (current) STL containers always assume that they manage their own memory, they can not be used in device code
  - We wrote a set of “device containers” that can interact with the data allocated by their “host container” counterparts



Data Types



- We tried to formalise the exchange of information between host and device code using explicit data types
  - Types that are able to “point at” data owned by some other object
- The “device containers” can then be constructed on top of an appropriate data object
- “View type” objects are trivially constructible/copyable, and can be passed directly to device kernels
  - But we also introduced memory managing data types, which can only be instantiated in host code

Device Code Example

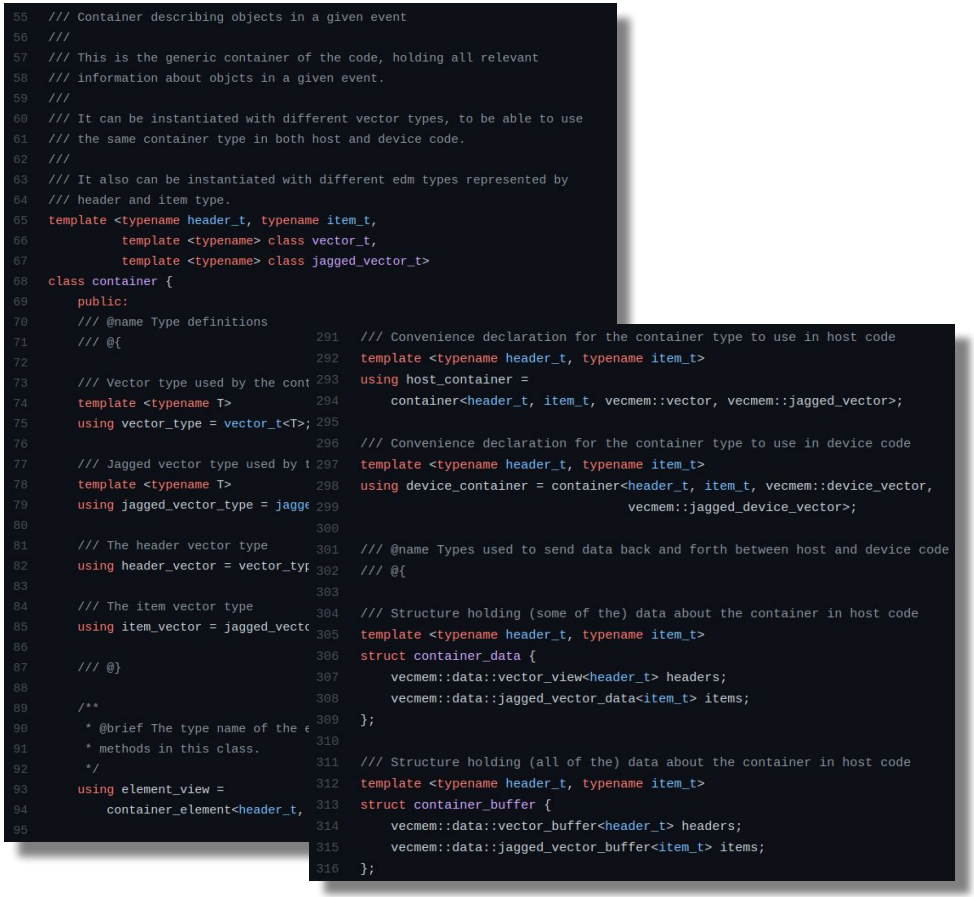
```
__global__
void linearTransformKernel(vecmem::vector_view<float> vec_data, float a, float b) {
   const std::size_t i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i >= vec_data.size()) {
      return;
   }
   vecmem::device_vector<float> vec(vec_data);
   vec.at(i) = vec[i] * a + b;
}

void linearTransform(vecmem::vector<float>& vec) {
   linearTransformKernel<<<vec.size(), 1>>>(vecmem::get_data(vec), 2.5f, 3.5f);
}
```

On-Device Container Modifications

- Apart from allowing convenient access to the elements of fixed sized containers, we also allow for appending elements to resizable vectors
  - This only works with separate host/device memory management, as we can not change the memory under an std::vector's feet
- In our current algorithms this comes in very handy in many situations

```
__global__
void fillVector(
  vecmem::data::vector_view<float> vec_data) {
  …
  vecmem::device_vector<float> vec(vec_data);
  vec.push_back(<foo>);
  …
}

{
  vecmem::cuda::host_memory_resource host_mr;
  vecmem::cuda::device_memory_resource device_mr;
  vecmem::cuda::copy copy;

  vecmem::data::vector_buffer<float>
    device_buffer(MAX_SIZE, 0, device_mr);
  copy.setup(device_buffer);

  fillVector<<<…>>>(device_buffer);

  vecmem::vector<float> host_vector;
  copy(device_buffer, host_vector);
}
```

Complex Types/Containers



```
/// Container describing objects in a given event
///
/// This is the generic container of the code, holding all relevant
/// information about objcts in a given event.
///
/// It can be instantiated with different vector types, to be able to use
/// the same container type in both host and device code.
///
/// It also can be instantiated with different edm types represented by
/// header and item type.
template <typename header_t, typename item_t,
          template <typename> class vector_t,
          template <typename> class jagged_vector_t>
class container {
    public:
    /// @name Type definitions
    /// @{

    /// Vector type used by the cont
    template <typename T>
    using vector_type = vector_t<T>;

    /// Jagged vector type used by t
    template <typename T>
    using jagged_vector_type = jagge

    /// The header vector type
    using header_vector = vector_typ

    /// The item vector type
    using item_vector = jagged_vecto

    /// @}

    /**
     * @brief The type name of the e
     * methods in this class.
     */
    using element_view =
        container_element<header_t,
```

```
/// Convenience declaration for the container type to use in host code
template <typename header_t, typename item_t>
using host_container =
    container<header_t, item_t, vecmem::vector, vecmem::jagged_vector>;

/// Convenience declaration for the container type to use in device code
template <typename header_t, typename item_t>
using device_container = container<header_t, item_t, vecmem::device_vector,
                                   vecmem::jagged_device_vector>;

/// @name Types used to send data back and forth between host and device code
/// @{

/// Structure holding (some of the) data about the container in host code
template <typename header_t, typename item_t>
struct container_data {
    vecmem::data::vector_view<header_t> headers;
    vecmem::data::jagged_vector_data<item_t> items;
};

/// Structure holding (all of the) data about the container in host code
template <typename header_t, typename item_t>
struct container_buffer {
    vecmem::data::vector_buffer<header_t> headers;
    vecmem::data::jagged_vector_buffer<item_t> items;
};
```

- In detray and traccc we define some types that we want to use both as “host” and as “device” objects / containers
  - Instead of duplicating definitions for the different vector types, we made these data types templated on the vector type that they should use
  - Allowing us to write even algorithmic code in a way that it could be used directly on top of host vectors, or in device code on device vectors

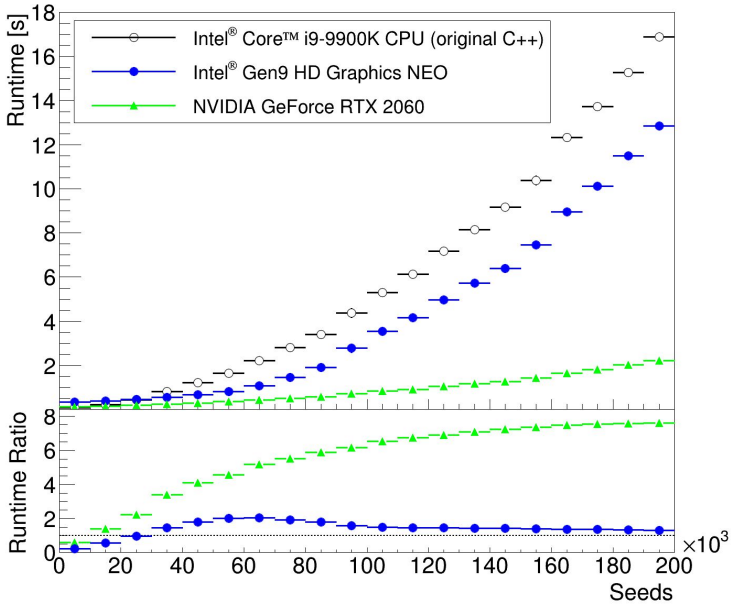
Miscellaneous Types

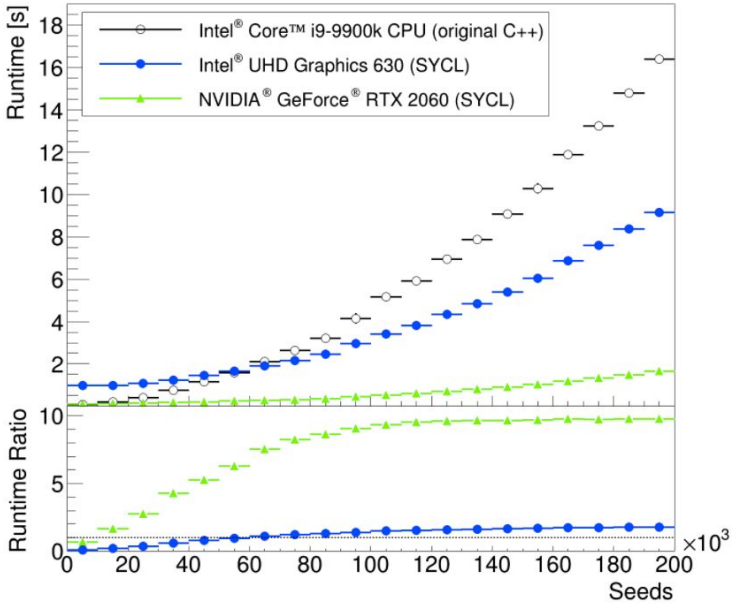
- We have also defined a number of other, hopefully useful types
  - Containers with a compile-time (maximum) size
  - vecmem::allocator for “non-container” memory allocations
  - vecmem::atomic for abstracting atomic operations for all supported languages
  - vecmem::memory_monitor for collecting simple information about an application’s behaviour
- Mainly driven by what we could think of, and what we ended up needing for the tracking demonstrator

```
namespace vecmem {

/// Class mimicking @c std::vector on top of a fixed sized array
///
/// This can come in handy when needing vector arithmetics in device code,
/// without resorting to heap allocations.
///
/// The type does come with a significant limitation over @c std::vector.
/// It has a maximal/fixed size that needs to be chosen at compile time.
///
template <typename TYPE, std::size_t MAX_SIZE>
class static_vector {

public:
    /// @name Type definitions
    /// @{

    /// Type of the array elem
    typedef TYPE value_type;
    /// Size type for the arra
    typedef std::size_t size_t
    /// Pointer difference typ
    typedef std::ptrdiff_t dif

    /// The maximal size of th
    static constexpr size_type
    /// The size of the vector
    static constexpr size_type
    /// Type of the array hold
    typedef
        typename details::arra
            array_type;

    /// Value reference type
    typedef value_type& refere
    /// Constant value referen
    typedef const value_type&
    /// Value pointer type
    typedef value_type* pointe
    /// Constant value pointer
    typedef const value_type*
```

```
namespace vecmem {

/// Class collecting some basic set of memory allocation statistics
///
/// Objects of this class can be used together with
/// @c vecmem::instrumenting_memory_resource to easily access a common set of
/// useful performance metrics about an application.
///
/// Note that the lifetime of this object must be at least as long as the
/// lifetime of the connected memory resource!
///
class VECMEM_CORE_EXPORT memory_monitor {

public:
    /// Constructor with a memory resource reference
    memory_monitor(instrumenting_memory_resource& resource);

    /// Get the total amount of allocations
    std::size_t total_allocation() const;
    /// Get the outstanding allocation left after all operations
    std::size_t outstanding_allocation() const;
    /// Get the average allocation size
    std::size_t average_allocation() const;
    /// Get the maximal concurrent allocation
    std::size_t maximal_allocation() const;
```

Memory Caching in Action





Summary

- A lot of interesting features are coming in C++2X for GPU programming
  - We jumped on “memory resources” as something that is already available. But std::atomic_ref, std::mdspan and std::execution will all be very relevant for HEP code. We should be mindful of these upcoming language features when writing our accelerated code already today.
- We are still very much doing R&D with this code in the context of the ATLAS GPU tracking demonstrator
  - With the goal of merging this kind of memory management into ATLAS’s offline software before Run-4
- If you are at all interested in giving the code a try, don’t be shy to get in contact with us
  - Hopefully next year we will have some news about interfacing LLAMA with VecMem as well 😉

http://home.cern