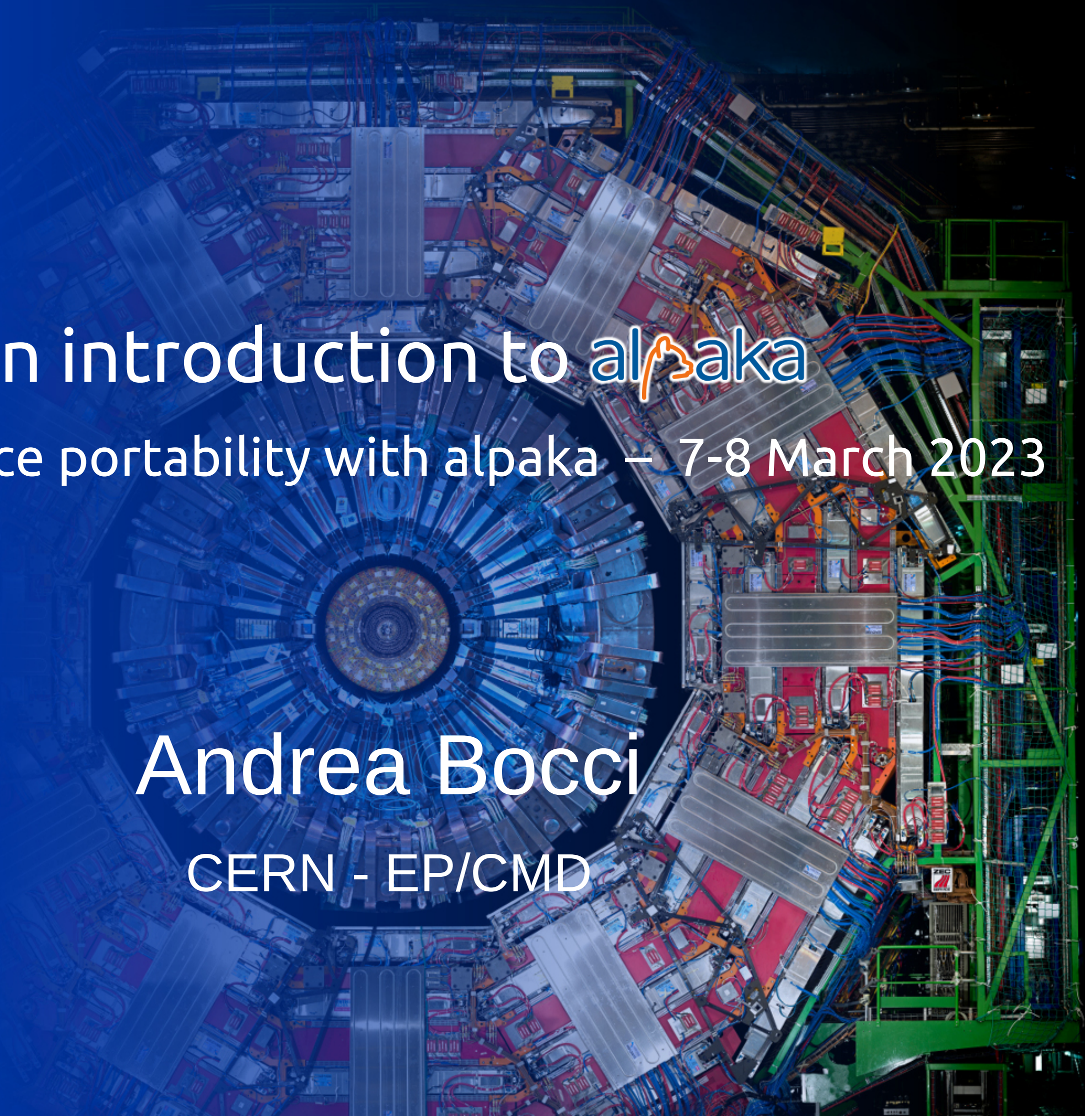
An introduction to alpaka

performance portability with alpaka – 7-8 March 2023

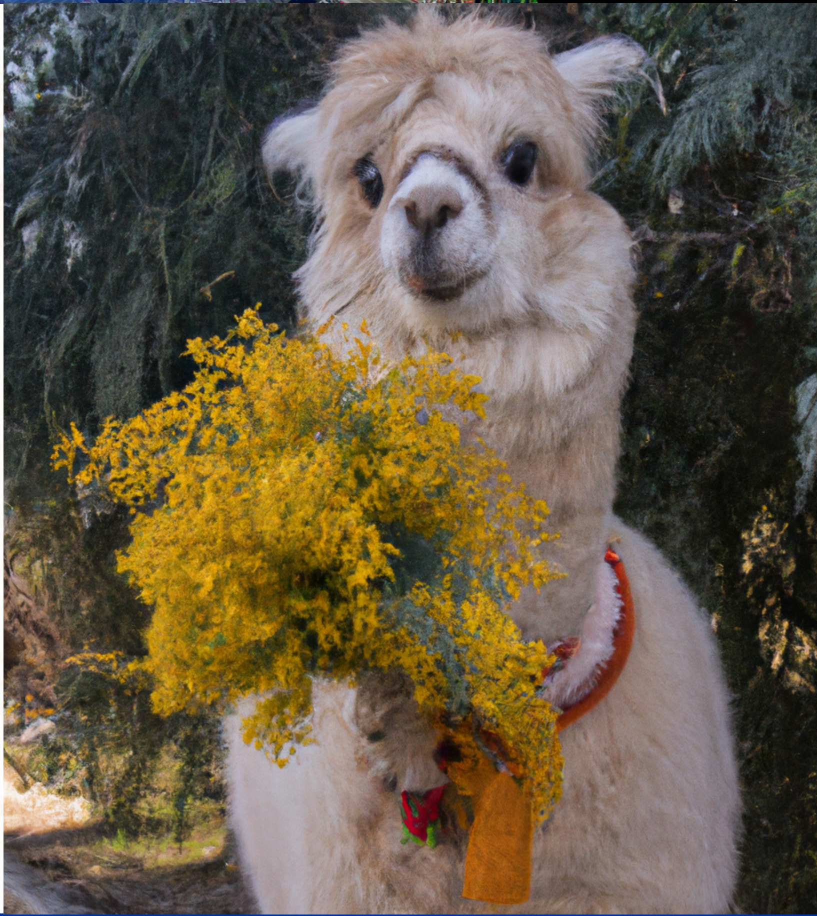
Andrea Bocci

CERN - EP/CMD



overview

- yesterday we have seen
  - what *performance portability* means and discovered the Alpaka library
  - how to set up Alpaka for a simple project
  - how to compile a single source file for different back-ends
  - what are Alpaka platforms, devices, queues and events
- today we will learn
  - how to work with host and device memory
  - how to write device functions and kernels
  - how to use an Alpaka accelerator and work division to launch a kernel
  - and see a complete example !



memory operations

Buffers and Views

- can refer to memory on the host or on any device
  - general purpose host memory (e.g. as returned by `malloc` or `new`)
  - pinned host memory, visible by devices on a given platform (e.g. as returned by `cudaMallocHost`)
  - global device memory (e.g. as returned by `cudaMalloc`)
- can have arbitrary dimensions
- 0-dimensional buffers and views wrap and provide access to a single element:

```
float x = *buffer;
float y = buffer->pt();
```

- 1-dimensional buffers and views wrap and provide access to an array of elements:

```
float x = buffer[i];
```

- N-dimensional buffers and views wrap arbitrary memory areas:

```
float* p = std::data(buffer);
```

  - expect a nicer accessor syntax with c++23 `std::mdspan` and improved `operator[]`

memory buffers

- buffers *own* the memory they point to
  - a host memory buffer can use either standard host memory, or pinned host memory mapped to be visible by the GPUs in a given platform
  - a buffer knows what device the memory is on, and how to free it
- buffers have shared ownership of the memory
  - like `shared_ptr<T>`
  - making a copy of a buffer creates a second handle to the same underlying memory
  - the memory is automatically freed when the last buffer object is destroyed (*e.g.* goes out of scope)
    - with queue-ordered semantic, memory is freed when the work submitted to the queue associate to the buffer is complete
- note that buffers always allow modifying their content
  - a `Buffer<const T>` would not be useful, because its contents could never be set
  - a `const Buffer<T>` does not prevent changes to the contents, as they can be modified through a copy

allocating memory

- buffer allocations and deallocations can be immediate or queue-ordered
  - immediate operations
    - allocate and free the memory immediately
    - may result in a device-wide synchronisation
    - *e.g.* `malloc` / `free` or `cudaMalloc` / `cudaFree`

```
// allocate an array of "size" floats in standard host memory
auto buffer = alpaka::allocBuf<float, uint32_t>(host, size);

// allocate an array of "size" floats in pinned host memory
// mapped to be efficiently copieable to/from all the devices on the Platform
auto buffer = alpaka::allocMappedBuf<Platform, float, uint32_t>(host, size);

// alloca an array of "size" floats in global device memory
auto buffer = alpaka::allocBuf<float, uint32_t>(device, size);
```

  - queue-ordered operations are usually asynchronous, and may cache allocations
    - guarantee that the memory is allocated before any further operations submitted to the queue are executed
    - guarantee that the memory will be freed once all pending operation in the queue are complete
    - *e.g.* `cudaMallocAsync` / `cudaFreeAsync`

```
// allocate an array of "size" floats in global gpu memory, ordered along queue
auto buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, size);
```

    - available only on device that support it (CPUs, NVIDIA CUDA ≥ 11.2, AMD ROCm ≥ 5.4)

using buffers

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/03_memory.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
auto host_buffer =
    alpaka::allocMappedBuf<Platform, float, uint32_t>(host, Vec1D{size});
std::cout << "pinned host memory buffer at " << std::data(host_buffer) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_buffer[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

```cpp
  // ...

{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, device_buffer);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_buffer[i] << ' ';
}
std::cout << '\n';
```

using buffers

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/03_memory.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
auto host_buffer =
    alpaka::allocMappedBuf<Platform, float, uint32_t>(host, Vec1D{size});
std::cout << "pinned host memory buffer at " << std::data(host_buffer) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_buffer[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

```cpp
  // ...

{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, device_buffer);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_buffer[i] << ' ';
}
std::cout << '\n';
```

allocate buffers

using buffers

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/03_memory.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
auto host_buffer =
    alpaka::allocMappedBuf<Platform, float, uint32_t>(host, Vec1D{size});
std::cout << "pinned host memory buffer at " << std::data(host_buffer) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_buffer[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

get the buffers' memory addresses

```cpp
// ...

{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, device_buffer);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_buffer[i] << ' ';
}
std::cout << '\n';
```

using buffers

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/03_memory.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
auto host_buffer =
    alpaka::allocMappedBuf<Platform, float, uint32_t>(host, Vec1D{size});
std::cout << "pinned host memory buffer at " << std::data(host_buffer) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_buffer[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

write to and read from the host buffer like a vector

```cpp
// ...

{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, device_buffer);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_buffer[i] << ' ';
}
std::cout << '\n';
```

using buffers

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/03_memory.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
auto host_buffer =
    alpaka::allocMappedBuf<Platform, float, uint32_t>(host, Vec1D{size});
std::cout << "pinned host memory buffer at " << std::data(host_buffer) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_buffer[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

memset and memcpy operations are always asynchronous

```cpp
  // ...

{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, device_buffer);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_buffer[i] << ' ';
}
std::cout << '\n';
```

memory views

- views wrap memory allocated by some other mechanism to provide a common interface
  - *e.g.* a local variable on the stack, or memory owned by an `std::vector`
  - views *do not own* the underlying memory
  - the lifetime of a view should not exceed that of the memory it points to

```
float* data = new float[size];
auto view = alpaka::ViewPlainPtr<float, uint32_t>(data, host, Vec1D{size});   // define a view for a C++ array
alpaka::memcpy(queue, view, device_buffer);                                     // copy the data to the array
```

- views to standard containers
  - Alpaka provides adaptors and can automatically use `std::array<T, N>` and `std::vector<T>` as views

```
std::vector<float> data(size);
alpaka::memcpy(queue, data, device_buffer);                                     // copy the data to the vector
```

- using views to emulate buffers to constant objects
  - buffers always allow modifying their content
  - but we can wrap them in a constant view: `alpaka::ViewConst<Buffer<T>>`

```
auto const_view = alpaka::ViewConst(device_buffer);
alpaka::memcpy(queue, host_buffer, const_view);                                 // copy the data to the host
```

using views

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/04_views.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
std::vector<float> host_data(size);
std::cout << "host vector at " << std::data(host_data) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_data[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

```cpp
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // create a read-only view to the device data
  auto const_view = alpaka::ViewConst(device_buffer);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, const_view);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_data[i] << ' ';
}
std::cout << '\n';
```

using views

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/04_views.cc

```
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
std::vector<float> host_data(size);
std::cout << "host vector at " << std::data(host_data) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_data[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

use a vector directly

```
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // create a read-only view to the device data
  auto const_view = alpaka::ViewConst(device_buffer);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, const_view);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_data[i] << ' ';
}
std::cout << '\n';
```

using views

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/04_views.cc

```cpp
// require at least one device
std::size_t n = alpaka::getDevCount<Platform>();
if (n == 0) {
  exit(EXIT_FAILURE);
}

// use the single host device
Host host = alpaka::getDevByIdx<HostPlatform>(0u);
std::cout << "Host:   " << alpaka::getName(host) << '\n';

// allocate a buffer of floats in host memory, mapped to ... the device
uint32_t size = 42;
std::vector<float> host_data(size);
std::cout << "host vector at " << std::data(host_data) << "\n\n";

// fill the host buffers with values
for (uint32_t i = 0; i < size; ++i) {
  host_data[i] = i;
}

// use the first device
Device device = alpaka::getDevByIdx<Platform>(0u);
std::cout << "Device: " << alpaka::getName(device) << '\n';

// create a work queue
Queue queue{device};
```

```cpp
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, Vec1D{size});
  std::cout << "memory buffer on " << alpaka::getName(alpaka::getDev(device_buffer))
            << " at " << std::data(device_buffer) << "\n\n";

  // set the device memory to all zeros (byte-wise, not element-wise)
  alpaka::memset(queue, device_buffer, 0x00);

  // create a read-only view to the device data
  auto const_view = alpaka::ViewConst(device_buffer);

  // copy the contents of the device buffer to the host buffer
  alpaka::memcpy(queue, host_buffer, const_view);

  // the device buffer goes out of scope, but the memory is freed only
  // once all enqueued operations have completed
}

// wait for all operations to complete
alpaka::wait(queue);

// read the content of the host buffer
for (uint32_t i = 0; i < size; ++i) {
  std::cout << host_data[i] << ' ';
}
std::cout << '\n';
```

copy from a const view
to garantee not
changing the device buffer

alpaka device API

device functions

- device functions are marked with the `ALPAKA_FN_ACC` macro

```
ALPAKA_FN_ACC
float my_func(float arg) { … }
```

- backend-specific functions
  - if the implementation of a device function may depend on the backend or on the work division into groups and threads, it should be templated on the Accelerator type, and take an Accelerator object

```
template <typename TAcc>
ALPAKA_FN_ACC
float my_func(TAcc const& acc, float arg) { … }
```

- the availability of C++ features depends on the backend and on the device compiler
  - dynamic memory allocation is (partially) supported, but strongly discouraged
  - c++ std containers should be avoid
  - exceptions are usually not supported
  - recursive functions are supported only by some backends (CUDA: yes, but often inefficient; SYCL: no)
  - c++20 is available in CUDA code only starting from CUDA 12.0
  - *etc.*

examples:

- mathematical operations are similar to what is available in the c++ standard:
  - *e.g.*
    `alpaka::math::sin(acc, arg)`
- atomic operations are similar to what is available in CUDA and HIP
  - *e.g.*
    `alpaka::atomicAdd(acc, T* address, T value, alpaka::hierarchy::Blocks)`
- warp-level functions are similar to what is available in CUDA and HIP
  - *e.g.*
    `alpaka::warp::ballot(acc, arg)`

kernels

- are implemented as an `ALPAKA_FN_ACC void operator()(…) const` function of a dedicated `struct` or `class`
  - kernels never return anything: `-> void`
  - kernels cannot change any data member on the host: must be declared `const`
- are always templated on the accelerator type, and take an accelerator object as the first argument

```
struct Kernel {
  template <typename TAcc>
  ALPAKA_FN_ACC void operator()(
    TAcc const& acc,
    float const* in1, float const* in2, float* out, size_t size) const
  {
    ...
  }
};
```

- the `TAcc acc` argument identifies the backend and provides the details of the work division

alpaka: grids, blocks, threads ...

- alpaka maintains the work division into blocks and threads used in CUDA and OpenCL:
  - a kernel launch is divided into a grid of **blocks**
    - the various block are scheduled independently, so they may be running concurrently or at different times
    - operations in different blocks cannot be synchronised
    - operations in different blocks can communicate only through the device global memory
  - each block is composed of **threads** running in parallel
    - threads in a block tend to run concurrently, but may diverge or be scheduled independently from each other
    - operations in a block can be synchronised, *e.g.* with `alpaka::syncBlockThreads(acc);`
    - operations in a block can communicate through shared memory
  - blocks can be decomposed into sub-groups, *i.e.* **warps**
    - threads in the same warp can synchronise and exchange data using more efficient primitives

… and elements ?

- to support efficient algorithms running on a CPU, alpaka introduces an additional level in the execution hierarchy: **elements**
  - each thread in a block may run on multiple consecutive elements
  - CPU backends usually run with multiple elements per thread
    - a good choice might be 16 elements, so 16 consecutive integers or floats can be loaded into a cache line
    - in principle, this could allow a host compiler to auto-vectorise the code, but more testing and development is needed !
  - GPU backends usually run with a single element per thread
    - memory accesses are already coalesced at the warp level
    - in principle, 2 elements per thread could be used with `short` or `float16` data

- kernel should be written to allow for different number of elements per thread
  - a common approach is to use
    - N blocks, M threads per block, 1 element per thread on a GPU
    - N blocks, 1 thread per block, M elements per thread on a CPU

a simple strided loop

- we provide a helper to implement a simple N-dimensional strided loop
  - the launch grid is tiled and repeated as many times as needed to cover the problem size
  - this tends to be the most efficient approach when all threads can work independently

```
#include "workdivision.h"

struct Kernel {
  template <typename TAcc>
  ALPAKA_FN_ACC void operator()(
    TAcc const& acc,
    float const* in1, float const* in2, float* out, size_t size) const
  {
    for (auto index : elements_with_stride(acc, size)) {
      out[index] = in1[index] + in2[index];
    }
  }
};
```

- for more complicated cases, use the `alpaka::getWorkDiv` and `alpaka::getIdx` functions

launching kernels

alpaka: work submission

`Accelerator`

- describes “how” a kernel runs on a device
  - N-dimensional work division (1D, 2D, 3D, …)
  - on the CPU, serial vs parallel execution at the thread and block level (single thread, multi-threads, TBB tasks, …)
  - implementation of shared memory, atomic operations, *etc.*
- accelerators are created only when a kernel is executed, and can only be accessed in device code
  - each device function can (should) be templated on the accelerator type, and take an accelerator as its first argument
  - the accelerator object can be used to extract the execution configuration (blocks, threads, elements)
  - the accelerator type can be used to implement per-accelerator behaviour
- for example, an algorithm can be implemented in device code using a parallel approach for a GPU-based accelerator, and a serial approach for a CPU-based accelerator

launching a kernel

- a kernel launch requires
  - the type of the accelerator where the kernel will run
  - the queue to submit the work to
  - the work division into blocks, threads, and elements
  - an instance of the type that implements the kernel
  - the arguments to the kernel function

- we provide some helper types and functions
  - `config.h` includes the aliases `Acc1D`, `Acc2D`, `Acc3D` for 1D, 2D and 3D kernels
  - `workdivision.h` provides the helper function `make_workdiv<TAcc>(blocks, threads_or_elements)`

```
// launch a 1-dimensional kernel with 32 groups of 32 threads (GPU) or elements (CPU)
auto grid = make_workdiv<Acc1D>(32, 32);
alpaka::exec<Acc1D>(queue, grid, Kernel{}, a.data(), b.data(), sum.data(), size);
```

a complete alpaka example

a complete alpaka example

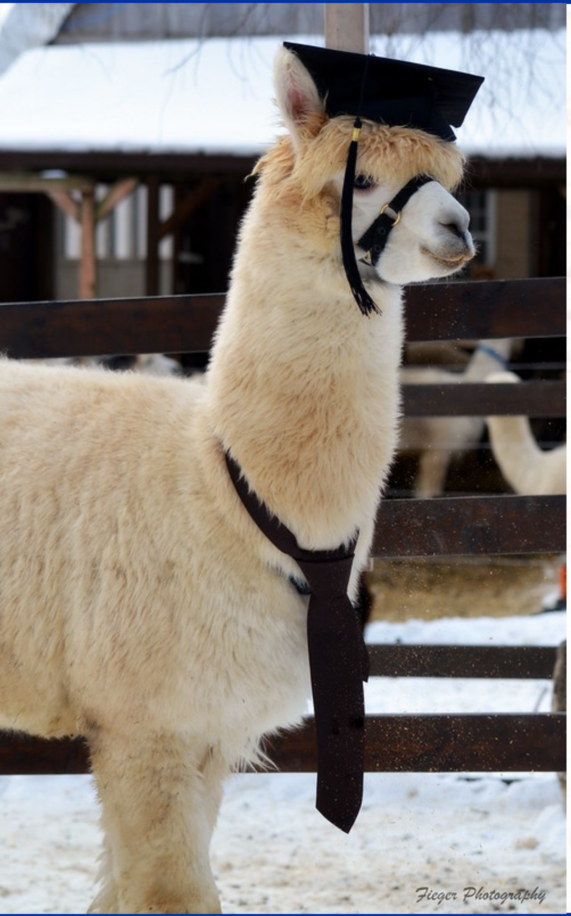
- running on the CPU

https://github.com/fwyzard/intro_to_alpaka/blob/master/alpaka/05_kernel.cc

```
$ ./05_kernel_cpu
Host:   AMD EPYC 7352 24-Core Processor
Device: AMD EPYC 7352 24-Core Processor
Testing VectorAddKernel with scalar indices with a grid of (32) blocks x (1) threads x (32) elements...
success
Testing VectorAddKernel1D with vector indices with a grid of (32) blocks x (1) threads x (32) elements...
success
Testing VectorAddKernel3D with vector indices with a grid of (5, 5, 1) blocks x (1, 1, 1) threads x (4, 4, 4) elements...
success
```

- running on the GPU

```
$ ./05_kernel_cuda
Host:   AMD EPYC 7352 24-Core Processor
Device: Tesla T4
Testing VectorAddKernel with scalar indices with a grid of (32) blocks x (32) threads x (1) elements...
success
Testing VectorAddKernel1D with vector indices with a grid of (32) blocks x (32) threads x (1) elements...
success
Testing VectorAddKernel3D with vector indices with a grid of (5, 5, 1) blocks x (4, 4, 4) threads x (1, 1, 1) elements...
success
```



- yesterday we learned
  - what *performance portability* means and discovered the Alpaka library
  - how to set up Alpaka for a simple project
  - how to compile a single source file for different back-ends
  - what are Alpaka platforms, devices, queues and events
- today we learned
  - how to work with host and device memory
  - how to write device functions and kernels
  - how to use an Alpaka accelerator and work division to launch a kernel
  - and see a complete example !
- congratulations!
  - now you can write *portable* and *performant* applications

(more) questions ?