Performance portability with alpaka

1st - 4th July 2025

Andrea Bocci

CERN - EP/CMD

last updated July 1st, 2025

who am I

- Dr. Andrea Bocci <andrea.bocci@cern.ch>, @fwyzard on CERN Mattermost
  - applied physicist working on the CMS experiment for over 20 years
  - at CERN since 2010
  - I've held various roles related to the High Level Trigger
    - started out as the b-tagging HLT contact
    - joined as (what today is called) HLT STORM convener
    - deputy Trigger Coordinator and Trigger Coordinator
    - HLT Upgrade convener, and editor for the DAQ and HLT Phase-2 TDR
    - currently, "GPU Trigger Officer"
  - for the last years, I've been working on GPUs and *performance portability*
    - together with a few colleagues at CERN and Fermilab
    - "Patatrack" pixel track and vertex reconstruction running on GPUs
    - R&D projects on CUDA, Alpaka, SYCL and Intel oneAPI
    - support for CUDA, HIP/ROCm, and Alpaka in CMSSW
    - Patatrack Hackathons !

performance portability

what is *portability* ?

- what do we mean by software *portability* ?
  - the possibility of running a software application or library on different platforms
    - different hardware architectures, different operating systems
    - e.g. Windows running on x86, OSX running on ARM, Linux running on RISC-V, *etc*.

- how do we achieve software *portability* ?
  - write software using a standardised language
    - C++, python, Java, *etc*.
  - use standard features
    - IEEE floating point numbers
  - use standard or portable libraries
    - C++ standard library, Boost, Eigen, *etc*.

portability: an example

- for example

portability/00_hello_world.cc

```
#include <cmath>
#include <cstdio>

void print_sqrt(double x) {
  printf("The square root of %g is %g\n", x, std::sqrt(x));
}

int main() {
  print_sqrt(2.);
}
```

should behave in the same way on all platforms that support a standard C++ compiler:

```
The square root of 2 is 1.41421
```

what about GPUs ?

- writing a program that offloads some of the computations to a GPU is somewhat different from writing a program that runs just on the CPU
  - inside a single application we have …
  - … different hardware architectures
  - … different memory spaces
  - … different way to call a function or launch a task
  - … different optimal algorithms
  - … different compilers
  - … different programming languages

- sometimes it may help to think about a GPU like programming a remote machine
  - compile for completely different targets
  - launching a kernel is similar to running a complete program

portability: the same example

portability/01_hello_world.cu

```
#include <cmath>
#include <cstdio>
#include <cuda_runtime.h>

__device__
void print_sqrt(double x) {
  printf("The square root of %g is %g\n", x, std::sqrt(x));
}

__global__
void kernel() {
  print_sqrt(2.);
}

int main() {
  kernel<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

```
The square root of 2 is 1.41421
```

portability: side by side

```cpp
#include <cmath>
#include <cstdio>

void print_sqrt(double x) {
  printf("The square root of %g is %g\n", x, std::sqrt(x));
}

int main() {
  print_sqrt(2.);
}
```

```
The square root of 2 is 1.41421
```

```cpp
#include <cmath>
#include <cstdio>
#include <cuda_runtime.h>

__device__
void print_sqrt(double x) {
  printf("The square root of %g is %g\n", x, std::sqrt(x));
}

__global__
void kernel() {
  print_sqrt(2.);
}

int main() {
  kernel<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

```
The square root of 2 is 1.41421
```

- we could
  - wrap the differences in a few macros or classes
  - share the common parts

so… are we done ?

- not really
  - trivially extending our example to an expensive computation would give horrible performance
- why ?
  - a CPU will run a single-threaded program very efficiently
  - a GPU will be heavily underutilised, using a single thread out of O(10k)
    - use only a small fraction of its computing power and memory bandwidth
    - loose any possibility of hiding memory latency, *etc.*
  - and what about different GPU back-ends ?
- what we need is *performance portability*
  - write code in a way that can run on multiple platforms
  - leverage their potential
  - and achieve (almost) native performance on all of them

performance portability ?

OpenCL™

OpenMP®

AMD HIP

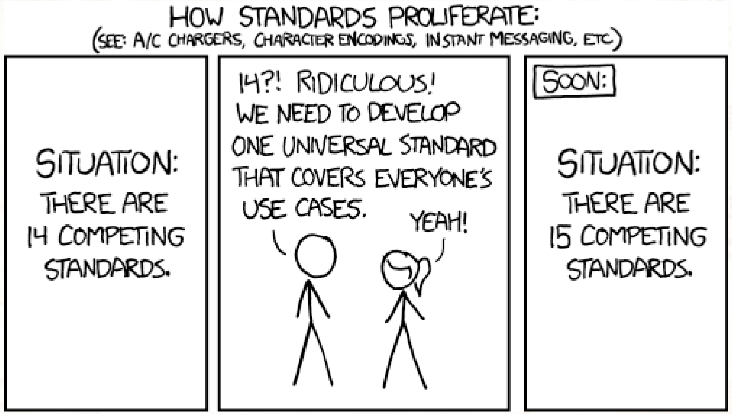
std::par

SYCL™

OpenACC

Kokkos

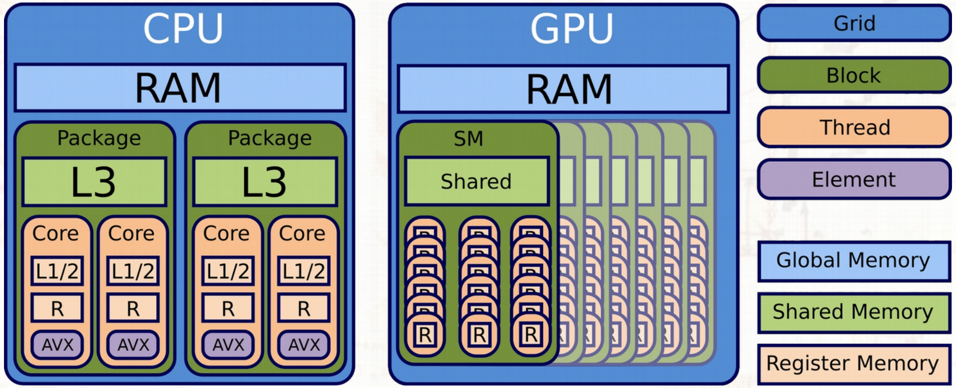
alpaka



the alpaka performance portability library

what is alpaka ?

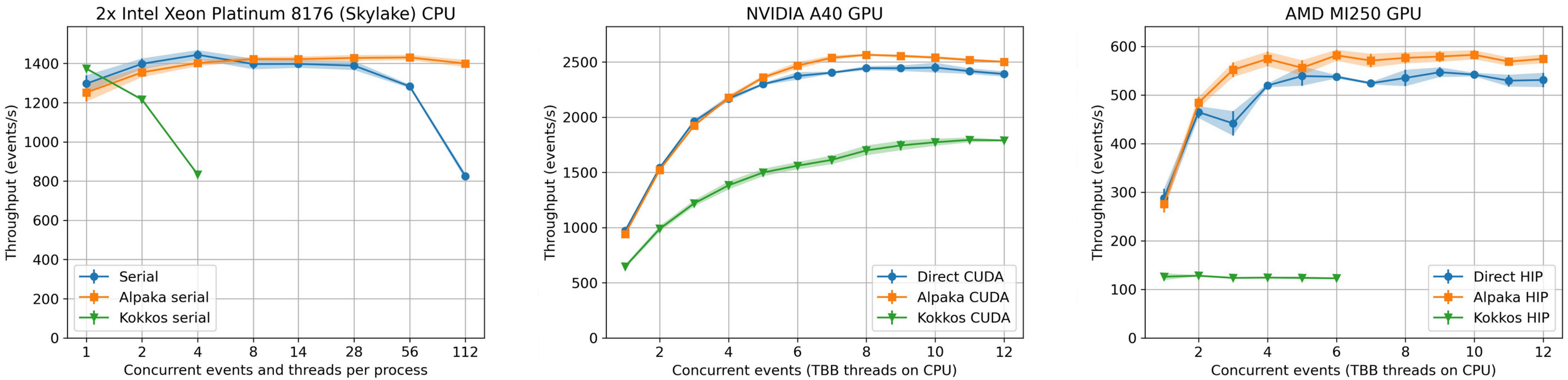
- alpaka is a header-only C++20 abstraction library for heterogeneous software development
  - it aims to provide *performance portability* through the abstraction of the underlying levels of parallelism
  - *may* expose the underlying details when necessary
  - (almost) *native* performance on different hardware



- supports all platforms of interest for HEP
  - x86 and ARM CPUs
    - with serial and parallel execution
  - stable support for NVIDIA and AMD GPUs
    - with CUDA and ROCm backends
  - experimental support for Intel GPUs and Altera FPGAs, based on SYCL and oneAPI
- developed at CASUS at HZDR, and at CERN
  - open source project, easy to contribute to: https://github.com/alpaka-group/alpaka/
- it is production-ready today !
  - the latest documentation is available at https://alpaka.readthedocs.io/en/latest/index.html

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

why alpaka ?



studies done at **CERN** and ***HEP-CCE***

- support all platforms of interest to CMS with near-native performance
  - evaluated using as a benchmark the Patatrack pixeltrack-standalone demonstrator
- production ready in 2022-2023, with long term support and development plans

alpaka is under active development

- alpaka 1.0.0 released on November 2023
  - experimental support for Intel oneAPI, with SYCL Unified Shared Memory model
  - support c++23 `std::mdspan` and Kokkos' `mdspan`
- alpaka 1.1.0 released on January 2024 ← *used in CMS 2024 software releases*
  - stable support for Intel oneAPI
  - implement additional math functions and warp-level functions
- alpaka 1.2.0 released on October 2024 ← *used in CMS 2025 software releases*
  - more complete support for Intel oneAPI
  - introduce helpers for writing parallel kernels
- alpaka 1.3.0 released on June 2025
  - bug fix release for long term support, stable branch with support for c++17

alpaka is under active development

- alpaka 2.0.0 released on June 2025
  - move to c++20 and introduce Concepts
  - make more device-side operations `constexpr`
  - improve memory buffers and views, support for “constant buffers”
  - improve support for Intel oneAPI and Altera FPGA
- under development:
  - support grid-wide synchronisation
  - support unified memory
  - support CUDA graphs / HIP graphs / TBB flow graphs

alpaka core concepts

Host-side API

- initialisation and device selection: `Platforms` and `Devices`
- asynchronous operations and synchronisation: `Queues` and `Events`
- owning memory `Buffers` and non-owning memory `Views`
- submitting work to devices: work division and `Accelerators`

Device-side API

- plain C++ for device functions and kernels
- shared memory, atomic operations, and memory fences
- primitives for mathematical operations
- warp-level primitives for synchronisation and data exchange *(not covered)*
- random number generator *(not covered)*

nota bene:

- most alpaka API objects behave like `shared_ptr`s, and should be passed by value or by reference to const (*i.e.* `const&`)

platforms and devices

`Platform`s and `Device`s

- identify the type of hardware (*e.g.* host CPUs or NVIDIA GPUs) and individual devices (*e.g.* each single GPU) present on the machine
- the CPU device `DevCpu` serves two purposes:
  - as the “host” device, for managing the data flow (*e.g.* perform memory allocation and transfers, launch kernels, *etc.*)
  - as an “accelerator” device, for running heterogeneous code (*e.g.* to run an algorithm on the CPU)
- platforms and devices should be created at the start of the program and used consistently
  - may hold an internal state, avoid creating multiple instances for the same hardware
- some common cases

back end	alpaka platform	alpaka device
CPUs, serial or parallel	`PlatformCpu`	`DevCpu`
NVIDIA GPU, with CUDA	`PlatformCudaRt`	`DevCudaRt`
AMD GPUs, with HIP/ROCm	`PlatformHipRt`	`DevHipRt`

platforms and devices

- Alpaka provides a simple API to enumerate the devices on a given platform:

  - `alpaka::getDevCount(platform)`
    - returns the number of devices on the given `platform`

  - `alpaka::getDevByIdx(platform, index)`
    - initialises the `index`-th device on the `platform`, and returns the corresponding `Device` object

  - `alpaka::getDevs(platform)`
    - initialises all devices on the `platform`, and returns a `vector` of `Device` objects

  - `alpaka::getName(device)`
    - returns the name of the given device

your first alpaka application

alpaka/00_enumerate.cc

```
int main() {
  // the host abstraction always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << '\n';
  std::cout << std::endl;

  // get all the devices on the accelerator platform
  Platform platform;
  std::vector<Device> devices = alpaka::getDevs(platform);

  std::cout << "Accelerator platform: " << alpaka::core::demangled<Platform> << '\n';
  std::cout << "Found " << devices.size() << " device(s):\n";
  for (auto const& device : devices)
    std::cout << "  - " << alpaka::getName(device) << '\n';
  std::cout << std::endl;
}
```

your first alpaka application

alpaka/00_enumerate.cc

```
int main() {
  // the host abstraction always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << '\n';
  std::cout << std::endl;

  // get all the devices on the accelerator platform
  Platform platform;
  std::vector<Device> devices = alpaka::getDevs(platform);

  std::cout << "Accelerator platform: " << alpaka::core::demangled<Platform> << '\n';
  std::cout << "Found " << devices.size() << " device(s):\n";
  for (auto const& device : devices)
    std::cout << "  - " << alpaka::getName(device) << '\n';
  std::cout << std::endl;
}
```

these are the *host* and *accelerator* platforms

your first alpaka application

alpaka/00_enumerate.cc

```
int main() {
  // the host abstraction always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << '\n';
  std::cout << std::endl;

  // get all the devices on the accelerator platform
  Platform platform;
  std::vector<Device> devices = alpaka::getDevs(platform);

  std::cout << "Accelerator platform: " << alpaka::core::demangled<Platform> << '\n';
  std::cout << "Found " << devices.size() << " device(s):\n";
  for (auto const& device : devices)
    std::cout << "  - " << alpaka::getName(device) << '\n';
  std::cout << std::endl;
}
```

`alpaka::core::demangled<T>` is a string with the "human readable" name of c++ type name

your first alpaka application

alpaka/00_enumerate.cc

```
int main() {
  // the host abstraction always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << '\n';
  std::cout << std::endl;

  // get all the devices on the accelerator platform
  Platform platform;
  std::vector<Device> devices = alpaka::getDevs(platform);

  std::cout << "Accelerator platform: " << alpaka::core::demangled<Platform> << '\n';
  std::cout << "Found " << devices.size() << " device(s):\n";
  for (auto const& device : devices)
    std::cout << "  - " << alpaka::getName(device) << '\n';
  std::cout << std::endl;
}
```

get the n^{th} device for the given platform

your first alpaka application

alpaka/00_enumerate.cc

```
int main() {
  // the host abstraction always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << '\n';
  std::cout << std::endl;

  // get all the devices on the accelerator platform
  Platform platform;
  std::vector<Device> devices = alpaka::getDevs(platform);

  std::cout << "Accelerator platform: " << alpaka::core::demangled<Platform> << '\n';
  std::cout << "Found " << devices.size() << " device(s):\n";
  for (auto const& device : devices)
    std::cout << "  - " << alpaka::getName(device) << '\n';
  std::cout << std::endl;
}
```

get all devices on the platform

your first alpaka application

alpaka/00_enumerate.cc

```
int main() {
  // the host abstraction always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << '\n';
  std::cout << std::endl;

  // get all the devices on the accelerator platform
  Platform platform;
  std::vector<Device> devices = alpaka::getDevs(platform);

  std::cout << "Accelerator platform: " << alpaka::core::demangled<Platform> << '\n';
  std::cout << "Found " << devices.size() << " device(s):\n";
  for (auto const& device : devices)
    std::cout << "  - " << alpaka::getName(device) << '\n';
  std::cout << std::endl;
}
```

get the name of each device

some important details

alpaka/00_enumerate.cc

```
#include <iostream>
#include <vector>

#include <alpaka/alpaka.hpp>

#include "config.h"

...
```

some important details

alpaka/00_enumerate.cc

```
#include <iostream>
#include <vector>

#include <alpaka/alpaka.hpp>

#include "config.h"

...
```

include the alpaka headers

some important details

alpaka/00_enumerate.cc

```
#include <iostream>
#include <vector>

#include <alpaka/alpaka.hpp>

#include "config.h"

...
```

include a header that defines the configuration for the various back-ends

let's build it …

- using the CPU as a single-threaded, serial "accelerator"
  - the CPU acts as both the "host" and the "device"
  - the application runs entirely on the CPU

```
g++ -std=c++20 -O2 -g \
    -I/opt/alpaka/include -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED \
    00_enumerate.cc \
    -o 00_enumerate_cpu
```

- using the CUDA GPUs as the "accelerator"
  - the CPU acts as the "host", the GPUs act as the "devices"
  - the application launches kernels that run on the GPUs

```
nvcc -x cu -expt-relaxed-constexpr -std=c++20 -O2 -g \
    -I/opt/alpaka/include -DALPAKA_ACC_GPU_CUDA_ENABLED \
    00_enumerate.cc \
    -o 00_enumerate_cuda
```

… and run it

```
$ ./00_enumerate_cpu
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformCpu
Found 1 device(s):
  - Intel Xeon Processor (Cascadelake)
```

```
$ ./00_enumerate_cuda
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Accelerator platform:
alpaka::PlatformUniformCudaHipRt<alpaka::ApiCudaRt>
Found 1 device(s):
  - Tesla V100-SXM2-32GB
```

how does it work ?

- Alpaka internally uses preprocessor symbols to enable the different backends:
  - `ALPAKA_ACC_GPU_CUDA_ENABLED` for running on NVIDIA GPUs
  - `ALPAKA_ACC_GPU_HIP_ENABLED` for running on AMD GPUs
  - `ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED` for running serially on a CPU
  - …

- in the first part of this tutorial we will build separate applications from each example
  - each application uses a single back-end
  - and is compiled with the corresponding compiler (`g++`, `nvcc`, `hipcc`, …)

- it is also possible to enable more than one back-end at a time
  - however, the underlying CUDA and HIP header files will clash, so one needs to use different "translation units" (compilation of a c++ file) for the different backends
  - and separate the host and device parts

where is the magic ?

alpaka/config.h

```
#if defined(ALPAKA_ACC_GPU_CUDA_ENABLED)
// CUDA backend
using Device = alpaka::DevCudaRt;
using Platform = alpaka::Platform<Device>;

#elif defined(ALPAKA_ACC_GPU_HIP_ENABLED)
// HIP/ROCm backend
using Device = alpaka::DevHipRt;
using Platform = alpaka::Platform<Device>;

#elif defined(ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED)
// CPU serial backend
using Device = alpaka::DevCpu;
using Platform = alpaka::Platform<Device>;

#else
// no backend specified
#error Please define one of ALPAKA_ACC_GPU_CUDA_ENABLED, ALPAKA_ACC_GPU_HIP_ENABLED, ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED

#endif
```

back end	alpaka platform	alpaka device
CPUs, serial or parallel	PlatformCpu	DevCpu
NVIDIA GPU, with CUDA	PlatformCudaRt	DevCudaRt
AMD GPUs, with HIP/ROCm	PlatformHipRt	DevHipRt

where is the magic ?

alpaka/config.h

```
#if defined(ALPAKA_ACC_GPU_CUDA_ENABLED)
// CUDA backend
using Device = alpaka::DevCudaRt;
using Platform = alpaka::Platform<Device>;

#elif defined(ALPAKA_ACC_GPU_HIP_ENABLED)
// HIP/ROCm backend
using Device = alpaka::DevHipRt;
using Platform = alpaka::Platform<Device>;

#elif defined(ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED)
// CPU serial backend
using Device = alpaka::DevCpu;
using Platform = alpaka::Platform<Device>;

#else
// no backend specified
#error Please define one of ALPAKA_ACC_GPU_CUDA_ENABLED, ALPAKA_ACC_GPU_HIP_ENABLED, ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED

#endif
```

depending on which back-end is enabled ...

where is the magic ?

alpaka/config.h

```
#if defined(ALPAKA_ACC_GPU_CUDA_ENABLED)
// CUDA backend
using Device = alpaka::DevCudaRt;
using Platform = alpaka::Platform<Device>;

#elif defined(ALPAKA_ACC_GPU_HIP_ENABLED)
// HIP/ROCm backend
using Device = alpaka::DevHipRt;
using Platform = alpaka::Platform<Device>;

#elif defined(ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED)
// CPU serial backend
using Device = alpaka::DevCpu;
using Platform = alpaka::Platform<Device>;

#else
// no backend specified
#error Please define one of ALPAKA_ACC_GPU_CUDA_ENABLED, ALPAKA_ACC_GPU_HIP_ENABLED, ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED

#endif
```

depending on which back-end is enabled, `Device` and `Platform` are aliased to different types

intermezzo: set up the examples

set up the examples

- get the examples from GitHub:

```
# clone the repository with the examples
$ git clone https://github.com/fwyzard/intro_to_alpaka.git -b bologna2025
$ cd intro_to_alpaka
$ ls -l
drwxr-xr-x. 2 abocci abocci  4096 Jul  1 11:14 alpaka
drwxr-xr-x. 2 abocci abocci  4096 Jul  1 11:14 enumerate
-rw-r--r--. 1 abocci abocci 11357 Jun 26 16:57 LICENSE
drwxr-xr-x. 2 abocci abocci   134 Jun 26 17:16 portability

$ cd alpaka/
$ make 00_enumerate_cpu
g++ -std=c++20 … -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED -o 00_enumerate.cc -o 00_enumerate_cpu

$ ./00_enumerate_cpu
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformCpu
Found 1 device(s):
  - Intel Xeon Processor (Cascadelake)
```

other back-ends

- automatically build for all architectures

```
$ make 00_enumerate
g++ -std=c++20 … -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED     00_enumerate.cc -o 00_enumerate_cpu
g++ -std=c++20 … -DALPAKA_ACC_CPU_B_SEQ_T_THREADS_ENABLED 00_enumerate.cc -o 00_enumerate_mt
g++ -std=c++20 … -DALPAKA_ACC_CPU_B_TBB_T_SEQ_ENABLED     00_enumerate.cc -o 00_enumerate_tbb
nvcc -x cu -std=c++20 … -DALPAKA_ACC_GPU_CUDA_ENABLED     00_enumerate.cc -o 00_enumerate_cuda

$ ./00_enumerate_cuda
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformUniformCudaHipRt<alpaka::ApiCudaRt>
Found 1 device(s):
  - Tesla V100-SXM2-32GB
```

queues and events

alpaka: asynchronous operations

Queues:

- identify a “work queue” where tasks (memory operations, kernel executions, …) are executed in order
  - for example, a queue could represent an underlying CUDA stream or a CPU thread
  - from the point of view of the host , queues can be synchronous or asynchronous
- with a synchronous (or *blocking*) queue:
  - any operation is executed immediately, before returning to the caller
  - the host automatically waits (blocks) until each operation is complete
- with an asynchronous (or *non-blocking*) queue:
  - any operation is executed in the background, and each call returns immediately, without waiting for its completion
  - the host needs to synchronize explicitly with the queue, before accessing the results of the operations
- in general, prefer using a synchronous queue on a CPU, and an asynchronous queue on a GPU
- queues are always associated to a specific device
- most alpaka operations (memory ops, kernel launches, *etc.*) are associated to a queue
- alpaka does not provide a “default queue”, create one explicitly

common operations on queues

- creating a queue of the predefined type associated to a device is as simple as
```
auto queue = Queue(device);
```
- waiting for all the asynchronous operations in a queue to complete is as simple as
```
alpaka::wait(queue);
```
- enqueue a host function
```
alpaka::enqueue(queue, host_function);
alpaka::enqueue(queue, [&]() { … });
```
- enqueue a device function (launch a kernel)
```
alpaka::exec<Acc>(queue, grid, kernel, args…);
```
- allocate, memset, fill, or copy memory host and device memory
```
auto buffer = alpaka::allocAsyncBuf<T, size_t>(queue, size);
alpaka::memset(queue, buffer, 0x00);
alpaka::fill(queue, buffer, value);
alpaka::memcpy(queue, destination, source);
```

Events:

- events identify points in time along a work queue
- can be used to query or wait for the readiness of a task submitted to a queue
- can be used to synchronise different queues
- like queues, events are always associated to a specific device

common operations on events

- events associated to a given device can be created with:

```
auto event = Event(device);
```

- events are enqueued to mark a given point along the queue:

```
alpaka::enqueue(queue, event);
```

  - an event is “complete” once all the work submitted to the queue before the event has been completed

- an event can be used to block the execution on the host until it is complete:

```
alpaka::wait(event);
```

  - blocks the execution on the host

- or to make an other queue wait until a given event (in a different queue) is complete:

```
alpaka::wait(other_queue, event);
```

  - does not block execution on the host
  - further work submitted to `other_queue` will only start after `event` is complete

- an event’s status can also be queried without blocking the execution:

```
alpaka::isComplete(event);
```

more magic

alpaka/config.h

```
#if defined(ALPAKA_ACC_GPU_CUDA_ENABLED)
// CUDA backend
using Queue = alpaka::Queue<Device, alpaka::NonBlocking>;
using Event = alpaka::Event<Queue>;

#elif defined(ALPAKA_ACC_GPU_HIP_ENABLED)
// HIP/ROCm backend
using Queue = alpaka::Queue<Device, alpaka::NonBlocking>;
using Event = alpaka::Event<Queue>;

#elif defined(ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED)
// CPU serial backend
using Queue = alpaka::Queue<Device, alpaka::Blocking>;
using Event = alpaka::Event<Queue>;

#else
// no backend specified
#error Please define one of ALPAKA_ACC_GPU_CUDA_ENABLED, ALPAKA_ACC_GPU_HIP_ENABLED, ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED

#endif
```

more magic

alpaka/config.h

```
#if defined(ALPAKA_ACC_GPU_CUDA_ENABLED)
// CUDA backend
using Queue = alpaka::Queue<Device, alpaka::NonBlocking>;
using Event = alpaka::Event<Queue>;

#elif defined(ALPAKA_ACC_GPU_HIP_ENABLED)
// HIP/ROCm backend
using Queue = alpaka::Queue<Device, alpaka::NonBlocking>;
using Event = alpaka::Event<Queue>;

#elif defined(ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED)
// CPU serial backend
using Queue = alpaka::Queue<Device, alpaka::Blocking>;
using Event = alpaka::Event<Queue>;

#else
// no backend specified
#error Please define one of ALPAKA_ACC_GPU_CUDA_ENABLED, ALPAKA_ACC_GPU_HIP_ENABLED, ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED

#endif
```

prefer asynchronous queues for a GPU

prefer synchronous queues for a CPU

playing with queues

alpaka/01_blocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::Blocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

playing with queues

alpaka/01_blocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::Blocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

we know this part

playing with queues

alpaka/01_blocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::Blocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

create a *blocking* queue on the host

playing with queues

alpaka/01_blocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::Blocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

this syntax introduces a *lambda expression* ...

playing with queues

alpaka/01_blocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::Blocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

this syntax introduces a *lambda expression* that performs these operations

togethwer with `alpaka::enqueue(...)`, this part

- creates an object that encapsulates some operations
- submits those opertations to run in a queue

playing with queues

alpaka/01_blocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::Blocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

wait for the enqueued operations to complete

let's build it and run it

- in this example we are not making use of any accelerator
  - let's build it only for the CPU back-end

```
$ make 01_blocking_queue_cpu
g++ -std=c++20 ... -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED 01_blocking_queue.cc
-o 01_blocking_queue_cpu
```

  - and run it

```
$ ./01_blocking_queue_cpu
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Enqueue some work
  - host task running...
  - host task complete
Wait for the enqueue work to complete...
All work has completed
```

an async example

alpaka/02_nonblocking_queue.cc

```cpp
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a non-blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::NonBlocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

an async example

alpaka/02_nonblocking_queue.cc

```
int main() {
  // the host platform always has a single device
  HostPlatform host_platform;
  Host host = alpaka::getDevByIdx(host_platform, 0u);

  std::cout << "Host platform: " << alpaka::core::demangled<HostPlatform> << '\n';
  std::cout << "Found 1 device:\n";
  std::cout << "  - " << alpaka::getName(host) << "\n\n";

  // create a non-blocking host queue and submit some work to it
  alpaka::Queue<Host, alpaka::NonBlocking> queue{host};

  std::cout << "Enqueue some work\n";
  alpaka::enqueue(queue, []() noexcept {
    std::cout << "  - host task running...\n";
    std::this_thread::sleep_for(std::chrono::seconds(5u));
    std::cout << "  - host task complete\n";
  });

  // wait for the work to complete
  std::cout << "Wait for the enqueue work to complete...\n";
  alpaka::wait(queue);
  std::cout << "All work has completed\n";
}
```

create a *non-blocking* queue on the host

let's build it and run it

- in this example, too, we are not making use of any accelerator
  - let's build it only for the CPU back-end – with POSIX threads

```
$ make 02_nonblocking_queue_cpu
g++ -std=c++20 ... -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED 02_nonblocking_queue.cc
-pthread -o 02_nonblocking_queue_cpu
```

  - and run it

```
$ ./02_nonblocking_queue_cpu
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Enqueue some work
Wait for the enqueue work to complete...
  - host task running...
  - host task complete
All work has completed
```

blocking vs non-blocking

```
$ ./01_blocking_queue_cpu
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Enqueue some work
  - host task running...
  - host task complete
Wait for the enqueue work to complete...
All work has completed
```

```
$ ./02_nonblocking_queue_cpu
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Enqueue some work
Wait for the enqueue work to complete...
  - host task running...
  - host task complete
All work has completed
```

- with a synchronous (or *blocking*) queue:
  - any operation is executed immediately, before returning to the caller
  - the host automatically waits (blocks) until each operation is complete
- with an asynchronous (or *non-blocking*) queue:
  - any operation is executed in the background, and each call returns immediately, without waiting for its completion
  - the host needs to synchronize explicitly with the queue, before accessing the results of the operations

memory operations

`Buffer`s and `View`s

- 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];
```

Buffers and Views

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

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

- we can use a nicer accessor syntax with c++23 `std::mdspan` and improved `operator[]`
  - alpaka can already use experimental `mdspan` support based on https://github.com/kokkos/mdspan

```
auto p = alpaka::experimental::getMdSpan(buffer);

// this syntax requires c++23
float f = p[i, j, k] = ...;

// this works with c++17 and later
float g = p(i, j, k);

// or, using an std::array as asingle index
std::array<int, 3> index{i, j, k};
float h = p[index];
```

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 *async* or *queue-ordered buffers*, memory is freed when the work submitted to the queue associated 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
    - alpaka 2.0 introduces `ConstBuffer<T>` objects, but support is still incomplete

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 copiable to/from all the devices on the platform
auto buffer = alpaka::allocMappedBuf<float, uint32_t>(host, platform, size);

// allocate 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

alpaka/03_memory.cc

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

// allocate a buffer of floats in pinned host memory
uint32_t size = 42;
auto host_buffer =
  alpaka::allocMappedBuf<float, uint32_t>(host, platform, 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;
}

// initialise the accelerator platform
Platform platform;
// 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, 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] << ' ';
}
```

using buffers

alpaka/03_memory.cc

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

// allocate a buffer of floats in pinned host memory
uint32_t size = 42;
auto host_buffer =
  alpaka::allocMappedBuf<float, uint32_t>(host, platform, 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;
}

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

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

allocate buffers

```cpp
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer =
    alpaka::allocAsyncBuf<float, uint32_t>(queue, 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] << ' ';
}
```

using buffers

alpaka/03_memory.cc

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

// allocate a buffer of floats in pinned host memory
uint32_t size = 42;
auto host_buffer =
  alpaka::allocMappedBuf<float, uint32_t>(host, platform, 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;
}

// initialise the accelerator platform
Platform platform;
// 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

```
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer =
    alpaka::allocAsyncBuf<float, uint32_t>(queue, 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] << ' ';
}
```

using buffers

alpaka/03_memory.cc

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

// allocate a buffer of floats in pinned host memory
uint32_t size = 42;
auto host_buffer =
  alpaka::allocMappedBuf<float, uint32_t>(host, platform, 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;
}

// initialise the accelerator platform
Platform platform;
// 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 or array

```
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer =
    alpaka::allocAsyncBuf<float, uint32_t>(queue, 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] << ' ';
}
```

using buffers

alpaka/03_memory.cc

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

// allocate a buffer of floats in pinned host memory
uint32_t size = 42;
auto host_buffer =
  alpaka::allocMappedBuf<float, uint32_t>(host, platform, 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;
}

// initialise the accelerator platform
Platform platform;
// 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, 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] << ' ';
}
```

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::createView(host, data, 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
  - we can wrap a buffer 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

alpaka/04_views.cc

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

// initialise the accelerator platform
Platform platform;

// allocate a buffer of floats in mapped host memory
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};
```

```
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, 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_data, 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] << ' '; }
```

using views

alpaka/04_views.cc

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

// initialise the accelerator platform
Platform platform;

// allocate a buffer of floats in mapped host memory
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 as an alpaka View

```cpp
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, 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_data, 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] << ' '; }
```

using views

alpaka/04_views.cc

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

// initialise the accelerator platform
Platform platform;

// allocate a buffer of floats in mapped host memory
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};
```

```
{
  // allocate a buffer of floats in global device memory, asynchronously
  auto device_buffer = alpaka::allocAsyncBuf<float, uint32_t>(queue, 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_data, 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] << ' '; }
```

pass a constant view to the copy operation 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, c++23 is not yet available
  - *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 back-end and provides the dimensionality and the work division

alpaka: grids, blocks, threads ...

- alpaka maintains the work division into blocks and threads used in CUDA, HIP 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** or **wavefronts**
    - 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 process 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
    - the goal is allow a host compiler to auto-vectorise the code, but more research and development is needed !
  - GPU backends usually run with a single element per thread
    - memory accesses are already coalesced at the warp level, but more writes per thread may improve the bandwidth
    - 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

- alpaka provides helper to implement a N-dimensional strided loops
  - the launch grid is tiled and repeated as many times as needed to cover the problem size
  - this is usually an efficient approach when all threads can work independently

```
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 : alpaka::uniformElements(acc, size)) {
      out[index] = in1[index] + in2[index];
    }
  }
};
```

  - also available for N-dimensional loops

```
    for (auto ndindex : alpaka::uniformElementsND(acc, {z,y,x})) { ... }
```

  - split across different dimensions, for non-uniform blocks, *etc.*
- for more complicated cases, use the `alpaka::getWorkDiv` and `alpaka::getIdx` functions

launching kernels

`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.*
- the `Accelerator` c++ type is available only when alpaka is being compiled for a specific back-end
  - the accelerator type can be used to specialise code and 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
- accelerator objects are created 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)

`Tag`

- identifies an `Accelerator` back-end, without the hardware and work division details
  - *e.g.* `TagCpuSerial`, `TagGpuCudaRt`, `TagGpuHipRt`, …
- unlike the `Accelerator`, the `Tag` C++ type is always available

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
  - `WorkDiv.hpp` provides the helper function `makeWorkDiv<TAcc>(blocks, threads_or_elements)`
    - taken from Alpaka tests

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

a complete alpaka example

a complete alpaka example

alpaka/05_kernel.cc

- running on the CPU

```
$ ./05_kernel_cpu
Host:   Intel Xeon Processor (Cascadelake)
Device: Intel Xeon Processor (Cascadelake)
Testing VectorAddKernel with scalar indices with a grid of (32) blocks x (1) threads x (32) elements...
success
```

- running on the GPU

```
$ ./05_kernel_cuda
Host:   Intel Xeon Processor (Cascadelake)
Device: Tesla V100-SXM2-32GB
Testing VectorAddKernel with scalar indices with a grid of (32) blocks x (32) threads x (1) elements...
success
```

a multidimensional alpaka example

alpaka/06_kernelnd.cc

- running on the CPU

```
$ ./06_kernelnd_cpu
Host:   Intel Xeon Processor (Cascadelake)
Device: Intel Xeon Processor (Cascadelake)
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

```
$ ./06_kernelnd_cuda
Host:   Intel Xeon Processor (Cascadelake)
Device: Tesla V100-SXM2-32GB
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
```

alpaka on different back-ends

parallel execution on CPUs

- parallel CPU back-end, using POSIX threads

```
$ make 06_kernelnd_mt
g++ -std=c++20 -O2 -g -I/opt/alpaka/include -DALPAKA_HAS_STD_ATOMIC_REF -pthread
-DALPAKA_ACC_CPU_B_SEQ_T_THREADS_ENABLED 06_kernelnd.cc -o 06_kernelnd_mt
```

```
$ ./06_kernelnd_mt
Host:   Intel Xeon Processor (Cascadelake)
Device: Intel Xeon Processor (Cascadelake)
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
```

parallel execution on CPUs

- parallel CPU back-end, using the Intel Threading Building Blocks library

```
$ make 06_kernelnd_tbb
g++ -std=c++20 -O2 -g -I/opt/alpaka/include -DALPAKA_HAS_STD_ATOMIC_REF -pthread
-I/opt/miniforge3/include -DALPAKA_ACC_CPU_B_TBB_T_SEQ_ENABLED 06_kernelnd.cc
-L/opt/miniforge3/lib -ltbb -o 06_kernelnd_tbb
```

```
$ ./06_kernelnd_tbb
Host:   Intel Xeon Processor (Cascadelake)
Device: Intel Xeon Processor (Cascadelake)
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
```

offloading to AMD GPUs

- AMD GPUs, using the HIP/ROCm runtime back-end

```
$ hipcc -std=c++20 -O2 -g -pthread \
    -I/opt/alpaka/include -DALPAKA_ACC_GPU_HIP_ENABLED \
    06_kernelnd.cc \
    -o 06_kernelnd_hip
```

```
$ ./06_kernelnd_hip
Host:   AMD EPYC 7A53 64-Core Processor
Device: AMD Instinct MI250X
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
```

Alpaka on the LUMI supercomputer !

offloading to Intel GPUs

- Intel GPUs, using the oneAPI back-end

```
$ icpx -fsycl -std=c++20 -O2 -g -pthread \
    -I/opt/alpaka/include -DALPAKA_ACC_SYCL_ENABLED -DALPAKA_SYCL_ONEAPI_GPU \
    06_kernelnd.cc \
    -o 06_kernelnd_sycl
```

```
$ ./06_kernelnd_sycl
Host:   Intel(R) Xeon(R) Platinum 8480+
Device: Intel(R) Data Center GPU Max 1100
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
```

Alpaka on the Aurora supercomputer ?

alpaka with mdspan

alpaka with mdspan

alpaka/07_mdspan.cc

- running on the CPU

```
 ./07_mdspan_cpu
Host:    Intel Xeon Processor (Cascadelake)
Device: Intel Xeon Processor (Cascadelake)
Testing VectorAddKernelMD with mdspan accessors with a grid of (5, 5, 1) blocks x (1, 1, 1) threads x (4, 4, 4)
elements...
success
```

- running on the GPU

```
$ ./07_mdspan_cuda
Host:    Intel Xeon Processor (Cascadelake)
Device: Tesla V100-SXM2-32GB
Testing VectorAddKernelMD with mdspan accessors with a grid of (5, 5, 1) blocks x (4, 4, 4) threads x (1, 1, 1)
elements...
success
```

a single alpaka application for multiple back-ends

a single application for multiple back-ends

- overall structure
  - config.h
    - defines different namespaces for each back-end
  - backend.h
    - provides a simple interface to the code in backend.cc
  - backend.cc
    - query the devices and accelerators
    - declares the code in a different namespace for each back-end
    - built N times as shared libraries, once for each back-end (CPU serial, CUDA, HIP, etc.)
  - main.cc
    - query the host part
    - links to the back-ends' shared libraries
    - call each back-end's implementation

enumerate/

```
enumerate/
├── Makefile
├── backend.cc
├── backend.h
├── config.h
├── enumerate
├── libbackend.cpu.so
├── libbackend.cuda.so
├── libbackend.hip.so
├── libbackend.mt.so
├── libbackend.tbb.so
└── main.o
```

a single application for multiple back-ends

```
$ ./enumerate
Host platform: alpaka::PlatformCpu
Found 1 device:
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformCpu
Found 1 device(s):
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformCpu
Found 1 device(s):
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformCpu
Found 1 device(s):
  - Intel Xeon Processor (Cascadelake)

Accelerator platform: alpaka::PlatformUniformCudaHipRt<alpaka::ApiCudaRt>
Found 1 device(s):
  - Tesla V100-SXM2-32GB
```

a single application for multiple back-ends

```
$ ./enumerate -v
...

Accelerator platform: alpaka::PlatformUniformCudaHipRt<alpaka::ApiCudaRt>
Found 1 device(s):
  - Tesla V100-SXM2-32GB
    - Accelerator name: alpaka::AccGpuUniformCudaHipRt<alpaka::ApiCudaRt,
std::integral_constant<long unsigned int, 3>, unsigned int>
        number of multi-processors:             80
        global memory free / total (bytes):     33748942848 / 34072559616
        shared memory per block (bytes):        49152
        max blocks per grid (z, y, x):          (65535, 65535, 2147483647)
        max threads per block (z, y, x):        (64, 1024, 1024)
        max elements per thread (z, y, x):      (4294967295, 4294967295, 4294967295)
        max number of blocks per grid:          4294967295
        max number of threads per block:        1024
        max number of elements per thread:      4294967295
        supported warp sizes:                   { 32 }
        preferred warp size:                    32
```

summary

summary

- during this tutorial 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
  - 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 worked up to a few complete examples
- congratulations!
  - now you can write *portable* and *performant* applications

(more) questions ?