Recap // Lecture 00

- alpaka is a C++ header-only library
- Abstraction library for parallel programming
- Single-source style
- Supports different back-ends for CPU and GPU programming
- Supports many modern compilers
- Supports different ecosystems
- Portable across operating systems
Recap // Lecture 00

- GitHub organization: [https://www.github.com/alpaka-group](https://www.github.com/alpaka-group)
- GitHub project: [https://www.github.com/alpaka-group/alpaka](https://www.github.com/alpaka-group/alpaka)
- Workshop slides: [https://www.github.com/alpaka-group/alpaka-workshop-slides](https://www.github.com/alpaka-group/alpaka-workshop-slides)
- Workshop examples: [https://www.github.com/alpaka-group/alpaka-workshop-examples](https://www.github.com/alpaka-group/alpaka-workshop-examples)
- Literature DOIs:
  - 10.1007/978-3-319-67630-2_36
  - 10.1109/IPDPSW.2016.50
  - 10.5281/zenodo.49768
Recap // Lecture 00

- Write algorithms once, run them everywhere!
- Decision on target platform made at compile time
- General case: Kernels are hardware-agnostic
- Special case: Kernels can be specialized for a concrete device / back-end
- Data parallelism achieved through a hierarchy of parallel threads
Recap // Lecture 00

• Workflow based on git and CMake 3.15+
• No core dependencies beside Boost
  • Depending on your back-end, additional dependencies may be required
• alpaka examples and test cases part of the source tree
• alpaka can be installed to a location of your choice
Recap

Lecture 10
Recap // Lecture 10

- `helloWorld` example
- `alpaka` spawns user-defined number of threads
- Threads may run in parallel
- Order of execution (and access to shared resources) unspecified

- `alpaka` Threads execute Kernels
- Threads are mapped to cores
- A set of cores is called Device
- Devices are attached to one Host
Recap // Lecture 10

- Kernel contains the algorithm
- Written on a per-data-element basis
- Kernels are functors (executable C++ struct / class)
  - `operator()` must be annotated with `ALPAKA_FN_ACC`
  - `operator()` must return `void`
  - `operator()` must be `const`
- A Thread applies a Kernel to a data element

```cpp
struct HelloWorldKernel {
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc) const {
        using namespace alpaka;
        uint32_t threadIdx = idx::getIdx<Grid, Threads>(acc)[0];
        printf("Hello, World from alpaka thread %u!\n", threadIdx);
    }
};
```
Recap // Lecture 10

- Number of Threads needs to fit the problem size
- Rule of thumb: One Thread per element, more Threads than cores
- Don’t launch too many Threads! Shared resources are scarce!

- All Threads form the Grid
- Grid is divided into Blocks of equal size
- Blocks have access to low-latency shared memory and Thread synchronization
- Grids and Blocks can be 1D, 2D or 3D
- alpaka API enables Grid navigation
Lecture 20
Recap // Lecture 20

• n-D work division
  • API functions for obtaining indices and extents
  • Index calculation
  • Beware of reversed index ordering!

• Computing π
  • Kernels accept three kinds of parameters: the accelerator, pointers to Device memory and and scalar values of trivially copyable types
  • Buffer iteration can be done through loops, Thread parallelism or a combination of both
  • alpaka Accelerator provides math functions
  • Rule of thumb: Launch one Thread per element, but this is not always ideal
  • Number of threads: $\text{blocksPerGrid} \times \text{threadsPerBlock}$
Recap // Lecture 20

- $n$-D Grid consists of all Threads
- Each Thread has a unique Grid index (accessible through alpaka's API)
- Threads are grouped into blocks of equal size
- Each thread has a unique Block-local index
- Threads inside Blocks have access to shared memory and Block-wide synchronization
• Changing the Accelerator is easy
  
  ```cpp
  using Acc = /* Accelerator of your choice */;
  ```

• Work division may need adaptation for new hardware type!

• Alpaka comes with a set of predefined Accelerators for CPUs and GPUs

• Hardware- and platform-specific details are abstracted away by the Accelerator

• Inside Kernel: Thread state, access to alpaka's device-side API

• On Host: Meta-parameter for choosing correct physical devices and dependent types
Recap // Lecture 30

- alpaka Devices represent physical devices
- Dependent on programmer’s Accelerator choice → only compatible Devices are detected by alpaka
- Devices enable physical device management and information

- Queues are used for communication between Host and Device
- They provide management for Device-side operations (Kernels, memory operations)
- Queues can be blocking or non-blocking
- Queues have different means of synchronization
Recap // Lecture 30

• Union of Accelerator, Device and Kernel is called Platform
• Portability is achieved through the Platform concept
• Do not make assumptions about the Device type in the Kernel! That is the Accelerator’s job.
• Know your Device type on the host and adapt the work division!
• By using multiple Accelerators at once, you can fully utilize heterogeneous systems

Write once, scale everywhere!