alpaka Parallel Programming Library In a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

alpaka – Abstraction Library for Parallel Kernel Acceleration

Alpaka is…

- A parallel programming library: Accelerate your code by exploiting your hardware‘s parallelism!
- An abstraction library: Create portable code that runs on CPUs and GPUs!
- Free & open-source software

alpaka in a Nutshell

Programming with alpaka

- C++ only!
- Header-only library: No additional runtime dependency introduced
- Modern library: alpaka is written entirely in C++17
- Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MS Visual Studio)
- Portable across operating systems: Linux, macOS, Windows are supported

alpaka In a nutshell

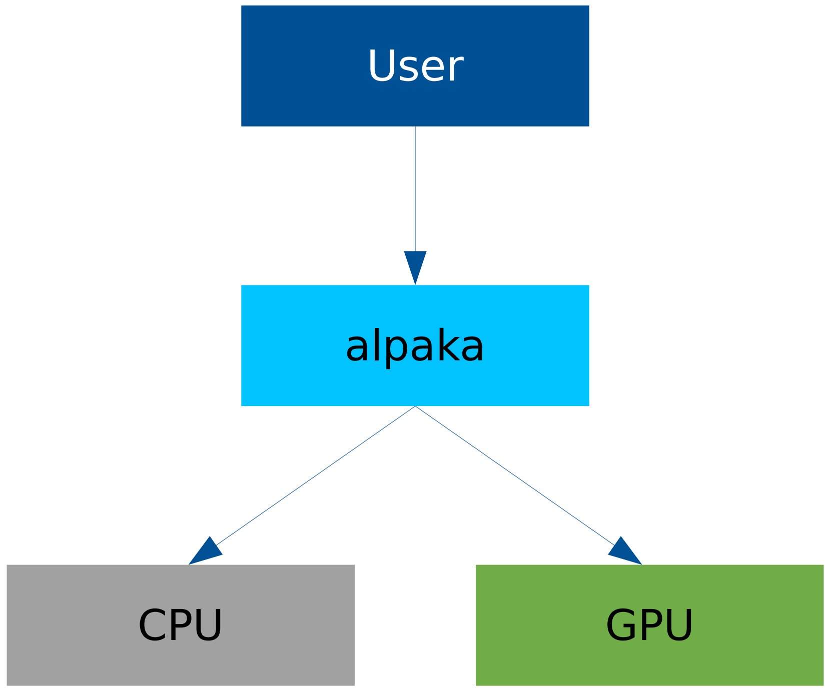
alpaka‘s purpose

Without alpaka

- Multiple hardware types commonly used (CPUs, GPUs, …)
- Increasingly heterogeneous hardware configurations available
- Platforms not inter-operable → parallel programs not easily portable

alpaka: one API to rule them all

- **Abstraction** (not hiding!) of the underlying hardware & software platforms
  - AMD, Nvidia, Intel GPUs, CPU
- **Easy change of the backend**
  - Code needs only minor adjustments to support different accelerators
- **Easy indexing of threads in kernels**
- **Easy setup of the type of parallelism** (Block sizes in grid, Thread sizes in block…)
- **Heterogenous Programming**: Using different backends in a synchronized manner



alpaka in a Nutshell

alpaka in the wild – example use case

PIConGPU: **https://github.com/ComputationalRadiationPhysics/picongpu**

- Fully relativistic, manycore, 3D3V particle-in-cell (PIC) code
- Central algorithm in plasma physics
- Scalable to more than 18,000 GPUs
- Developed at Helmholtz-Zentrum Dresden-Rossendorf

alpaka in a Nutshell

alpaka is free software (MPL 2.0). Find us on GitHub!

Our GitHub organization: https://www.github.com/alpaka-group

- Contains all alpaka-related projects, documentation, samples, …
- New contributors welcome!

The library: https://www.github.com/alpaka-group/alpaka

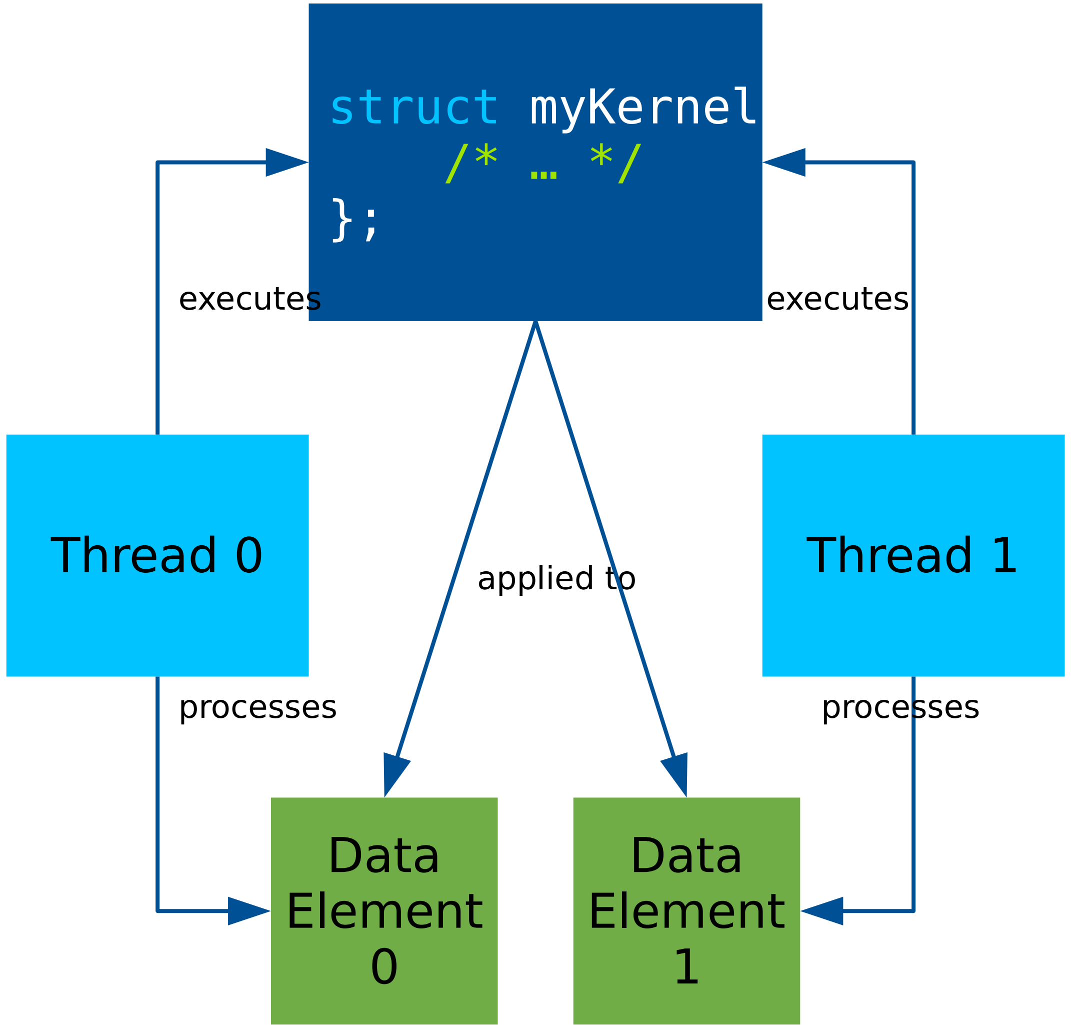
- Full source code
- Issue tracker
- Installation instructions
- Small examples

alpaka in a Nutshell

Basics: Threads and Kernels

- A Kernel is executed by a number of Threads
- Threads are executing the same algorithm for different data elements

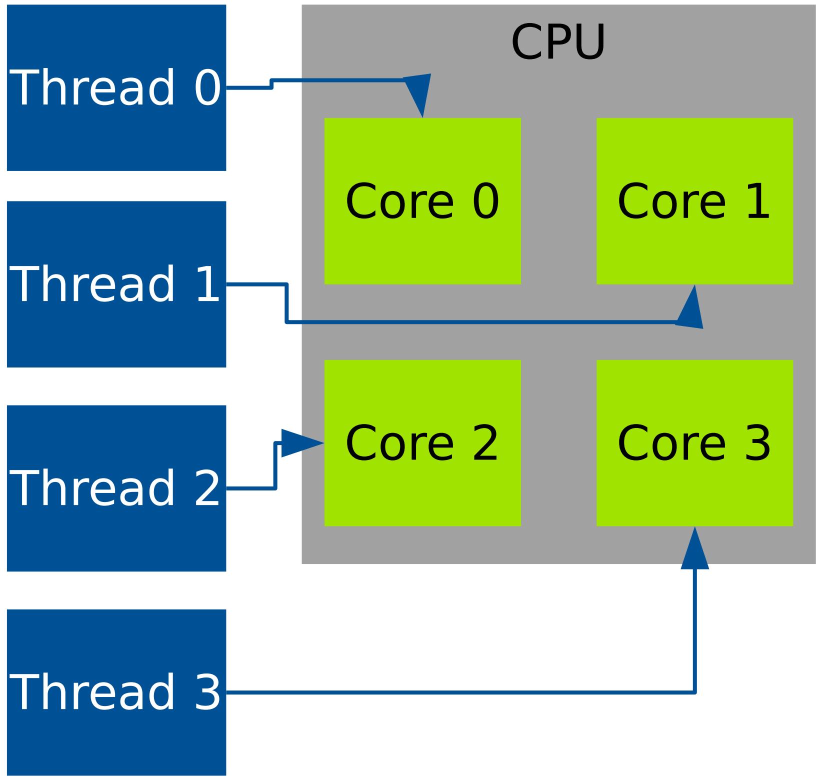
- A Kernel **defines** an algorithm
- A Thread **applies** an algorithm



alpaka in a Nutshell

Basics: Thread mapping on CPUs

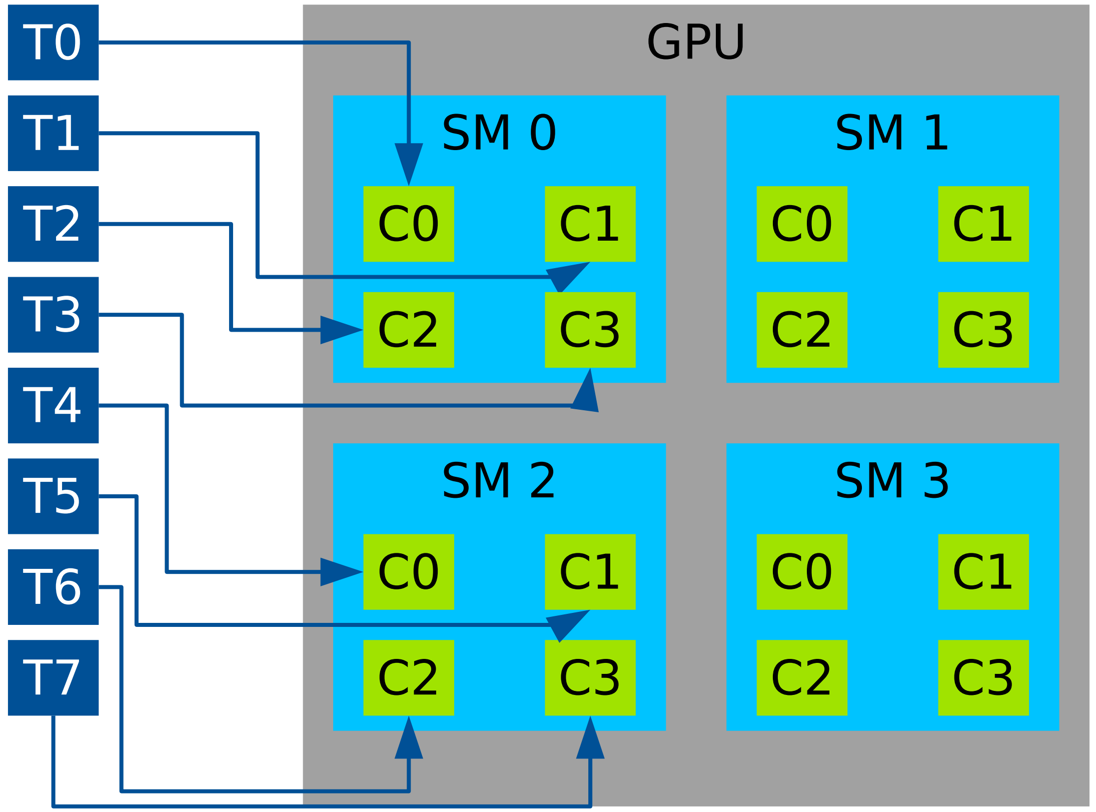
- CPU consists of multiple cores
  - Because of simultaneous multithreading there can be more logical than physical cores!
- alpaka Threads are executed by CPU cores



alpaka in a Nutshell

Basics: Thread mapping on GPUs

- GPU consists of streaming multiprocessors (SMs)
- Each SM consists of multiple cores
- alpaka Threads are executed by individual SM cores



alpaka in a Nutshell

Basics: What is an Alpaka Kernel?

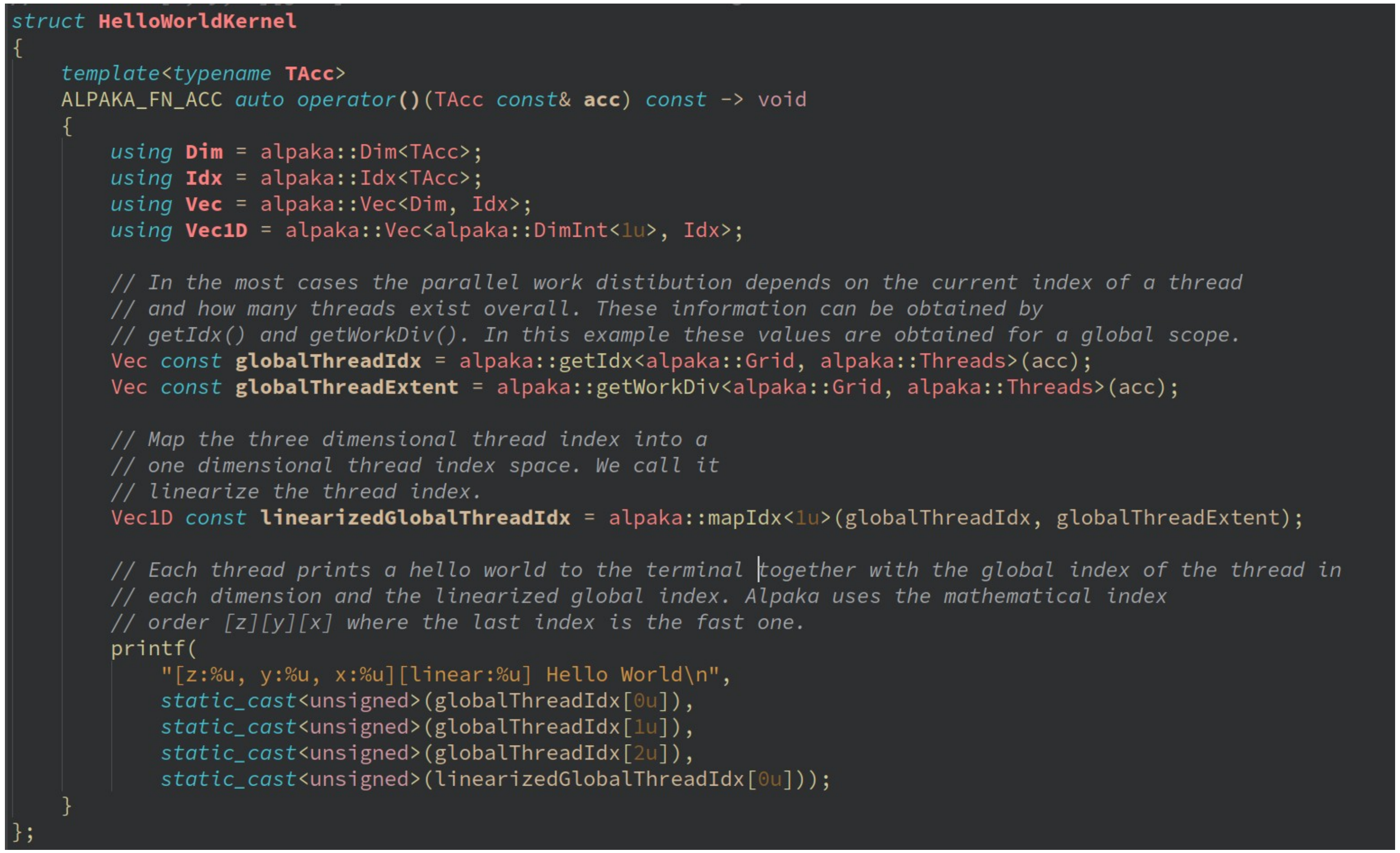
- Contains the algorithm
- Written on per-data-element basis
- alpaka Kernels are functors (function-like C++ structs / classes)
- `operator()` is annotated with `ALPAKA_FN_ACC` specifier
- `operator()` must return `void`
- `operator()` must be `const`

```
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);
    }
};
```

alpaka in a Nutshell

Why Alpaka - 1: Easy Indexing of Threads and Data

- Direct calculation of the index of a thread with respec to a grid or block origin in the kernel.
- Mapping the thread indexes to less dimensional space.



```
struct HelloWorldKernel
{
    template<typename TAcc>
    ALPAKA_FN_ACC auto operator()(TAcc const& acc) const -> void
    {
        using Dim = alpaka::Dim<TAcc>;
        using Idx = alpaka::Idx<TAcc>;
        using Vec = alpaka::Vec<Dim, Idx>;
        using Vec1D = alpaka::Vec<alpaka::DimInt<1u>, Idx>;

        // In the most cases the parallel work distibution depends on the current index of a thread
        // and how many threads exist overall. These information can be obtained by
        // getIdx() and getWorkDiv(). In this example these values are obtained for a global scope.
        Vec const globalThreadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc);
        Vec const globalThreadExtent = alpaka::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc);

        // Map the three dimensional thread index into a
        // one dimensional thread index space. We call it
        // linearize the thread index.
        Vec1D const linearizedGlobalThreadIdx = alpaka::mapIdx<1u>(globalThreadIdx, globalThreadExtent);

        // Each thread prints a hello world to the terminal together with the global index of the thread in
        // each dimension and the linearized global index. Alpaka uses the mathematical index
        // order [z][y][x] where the last index is the fast one.
        printf(
            "[z:%u, y:%u, x:%u][linear:%u] Hello World\n",
            static_cast<unsigned>(globalThreadIdx[0u]),
            static_cast<unsigned>(globalThreadIdx[1u]),
            static_cast<unsigned>(globalThreadIdx[2u]),
            static_cast<unsigned>(linearizedGlobalThreadIdx[0u]));
    }
};
```

alpaka in a Nutshell

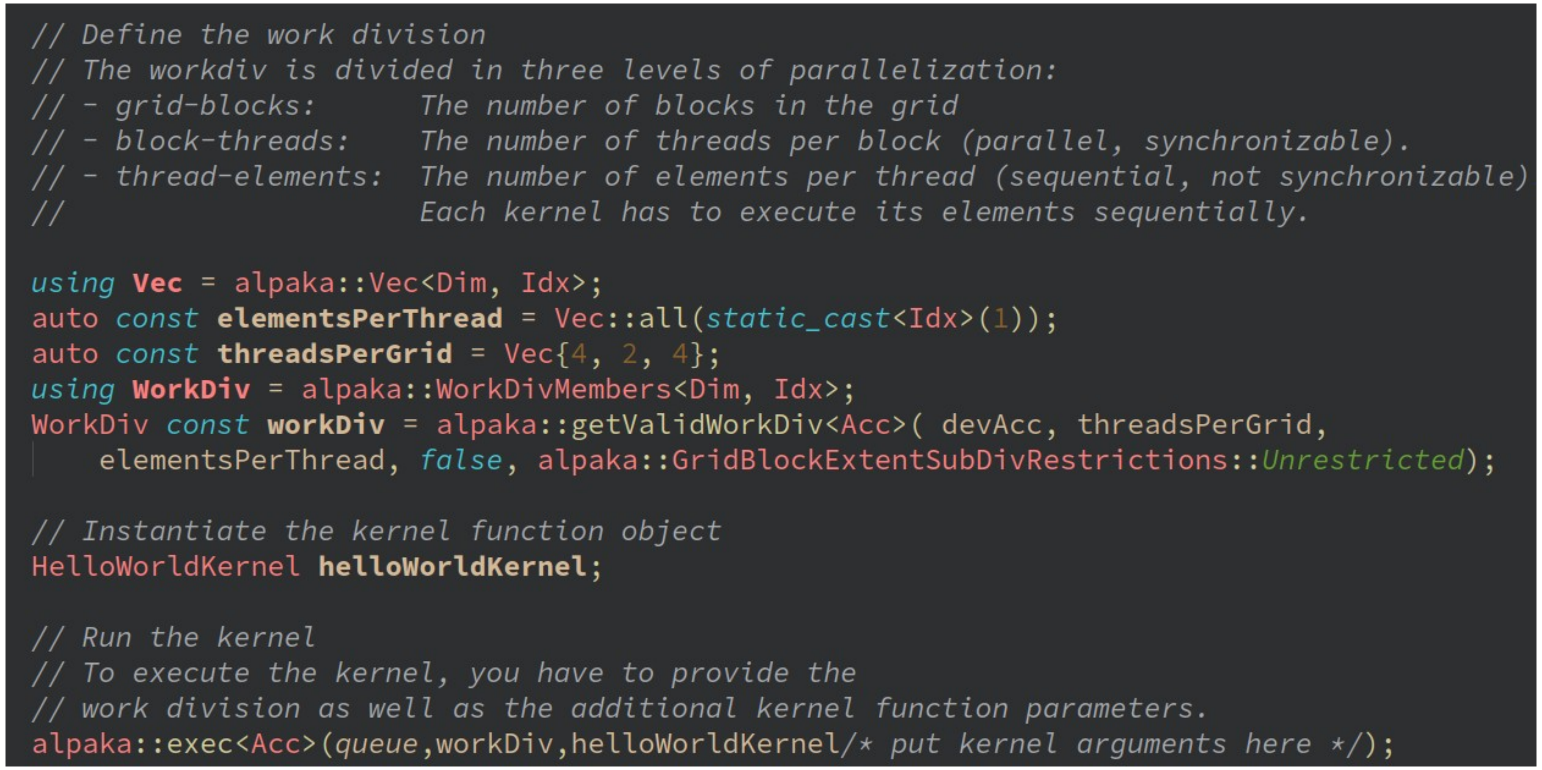
Obtaining the indices in Kernel

- alpaka provides several API functions for obtaining indices:
  - Index of Thread on the Grid: `idx::getIdx<alpaka::Grid, alpaka::Threads>(acc)[dim];`
  - Index of Thread on a Block: `idx::getIdx<alpaka::Block, alpaka::Threads>(acc)[dim];`
  - Index of Block on the Grid: `idx::getIdx<alpaka::Grid, alpaka::Blocks>(acc)[dim];`
- You can also obtain the extents of the Grid or the Blocks:
  - Number of Threads on the Grid: `workdiv::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc)[dim];`
  - Number of Threads on a Block: `workdiv::getWorkDiv<alpaka::Block, alpaka:Threads>(acc)[dim];`
  - Number of Blocks on the Grid: `workdiv::getWorkDiv<alpaka::Grid, alpaka::Blocks>(acc)[dim];`

alpaka in a Nutshell

Why Alpaka-2:
Easy definition of Type of Parallelism by WorkDivision

- Determines the number of kernel instantiations
- Determines the type of parallelism
  - Dimensions of a grid in terms of blocks,
  - Dimensions of a block in terms of threads
  - Elements per thread



```cpp
// Define the work division
// The workdiv is divided in three levels of parallelization:
// - grid-blocks:      The number of blocks in the grid
// - block-threads:    The number of threads per block (parallel, synchronizable).
// - thread-elements:  The number of elements per thread (sequential, not synchronizable)
//                     Each kernel has to execute its elements sequentially.

using Vec = alpaka::Vec<Dim, Idx>;
auto const elementsPerThread = Vec::all(static_cast<Idx>(1));
auto const threadsPerGrid = Vec{4, 2, 4};
using WorkDiv = alpaka::WorkDivMembers<Dim, Idx>;
WorkDiv const workDiv = alpaka::getValidWorkDiv<Acc>( devAcc, threadsPerGrid,
    elementsPerThread, false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

// Instantiate the kernel function object
HelloWorldKernel helloWorldKernel;

// Run the kernel
// To execute the kernel, you have to provide the
// work division as well as the additional kernel function parameters.
alpaka::exec<Acc>(queue,workDiv,helloWorldKernel/* put kernel arguments here */);
```

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Easy definition of Type of Parallelism: Preparing the Host for 2D Grid

- Go the top of `main()` and enable 2D dimensionality on the Host:

```
using Dim = dim::DimInt<2>;
```

- Further down in `main()`, set up a 2D Thread hierarchy:

```
auto blocksPerGrid = vec::Vec<Dim, Idx>{2u, 4u};
auto threadsPerBlock = vec::Vec<Dim, Idx>{1u, 1u};
auto elementsPerThread = vec::Vec<Dim, Idx>{1u, 1u};
```

alpaka in a nutshell: **Alpaka Structures**

Important Alpaka Structures

- **Accelerator** provides abstract view of all capable physical devices. `AccCpuThreads, AccGpuCudaRt, AccGpuHipRt...`
- **Device** represents a single physical device
- **Queue** enables communication between the host and a single Device
- **Platform** is a union of Accelerator, Device and Kernel
- **Task** is a device-side operation (e.g kernel, memory operation)
- Others: **Event**, **Buffer** (dynamic array), **Vector** (static array)

alpaka in a Nutshell: Accelerator

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Why Alpaka3: Changing the accelerator with minimal code change

- Accelerator concept is an abstraction of concrete devices and programming models
- The programmer changes the accelerator in just one line of code
- In the background, an entirely different code path for the "new" device is chosen
- Accelerator provides portable access to device-specific functions

```
/* Before the code change */
using Acc = acc::AccCpuOmp2Blocks<Dim, Idx>;


/* Kernels will run on CPUs */
/* Parallelism provided by OpenMP 2.x */


/* After the code change */

using Acc = acc::AccGpuHipRt<Dim, Idx>;


/* Kernels will run on AMD + NVIDIA GPUs */
/* Parallelism provided by HIP */
```

alpaka in a Nutshell: Accelerator

Switching the Accelerator

- alpaka provides a number of pre-defined Accelerators in the `acc` namespace.
- For GPUs:
  - `AccGpuCudaRt` for NVIDIA GPUs
  - `AccGpuHipRt` for AMD and NVIDIA GPUs
- For CPUs
  - `AccCpuFibers` based on Boost.fiber
  - `AccCpuOmp2Blocks` based on OpenMP 2.x
  - `AccCpuOmp4` based on OpenMP 4.x
  - `AccCpuTbbBlocks` based on TBB
  - `AccCpuThreads` based on `std::thread`

```
// Example: CPU accelerator

using Acc =
acc::AccCpuOmp2Blocks<Dim, Idx>;


// Example: CUDA GPU accelerator

using Acc = acc::AccGpuCudaRt<Dim,
Idx>;


// Example: HIP GPU accelerator

using Acc = acc::AccGpuHipRt<Dim,
Idx>;
```

alpaka in a Nutshell: Accelerator Details

- Accelerator chosen by the programmer and hides hardware specifics behind alpaka's abstract API

```
using Acc = acc::AccGpuCudaRt<Dim, Idx>;
```

- Accelerator is used on both Host and Device
- **Inside Kernel:** contains thread state, provides access to alpaka's device-side API

  The Accelerator provides the means to access to the indices

  ```
  // get thread index on the grid
  auto gridThreadIdx = idx::getIdx<Grid, Threads>(acc);
  ```

  ```
  // get block index on the grid
  auto gridBlockIdx = idx::getIdx<Grid, Blocks>(acc);
  ```

  - **The Accelerator gives access to alpaka's shared memory** (for threads inside the same block)

  ```
  // allocate a variable in block shared static memory
  auto & mySharedVar = block::shared::st::allocVar<int, __COUNTER__>(acc);
  ```

  ```
  // get pointer to the block shared dynamic memory
  float * mySharedBuffer = block::shared::dyn::getMem<float>(acc);
  ```

  - **It also enables synchronization on the block level**

  ```
  // synchronize all threads within the block
  block::sync::syncBlockThreads(acc);
  ```

  - **Internally, the accelerator maps all device-side functions to their native counterparts**
    - Device-side functions require the accelerator as first argument:
      - `math::sqrt(acc, /* … */);` `time::clock(acc);`
      - `atomic::atomicOp<atomic::op::Or>(acc, /* … */, hierarchy::Grids);` (Atomics)
      - `rand::distribution::createNormalReal<float>(acc);` (Random-number generation)
- **On Host:** Meta-parameter for choosing correct physical device and dependent types

alpaka in a Nutshell: Device

Physical device information and management by "alpaka Device"

- Each alpaka Device represents a single physical device;
- Contains device information:

```
auto const name = dev::getName(myDev);          // Back-end-defined device name
auto const bytes = dev::getMemBytes(myDev);     // Size of device memory
auto const free = dev::getFreeMemBytes(myDev); // Size of available device memory
```

- Provides the means for device management:

```
dev::reset(myDev);                              // Reset GPU device state
```

- Encapsulates back-end device:

```
auto nativeDevice = dev::getDev(myDev);         // nativeDevice is not portable!
```

alpaka in a Nutshell: Queue data structure

Queue: Connecting Host and Device

- alpaka Queues enable communication between Host and Device
- Two queue types: blocking and non-blocking
- Blocking queues block the Host until Device-side command returns
- Non-blocking queues return control to Host immediately, Device-side command runs asynchronously

```
// Choose queue behaviour - Blocking or NonBlocking
using QueueProperty = queue::NonBlocking;


// Define queue type

using Queue = queue::Queue<Acc, QueueProperty>;


// Create queue for communication with myDev
auto myQueue = Queue{myDev};
```

alpaka in a Nutshell : Queue data structure

Queue operations

- Queues execute Tasks (see next slide):

```
queue::enqueue(myQueue, taskRunKernel);
```

- Check for completion:

```
bool done = queue::empty(myQueue);
```

- Wait for completion, Events (see next slide), or other Queues:

```
wait::wait(myQueue);              // blocks caller until all operations have completed
wait::wait(myQueue, myEvent);     // blocks myQueue until myEvent has been reached
wait::wait(myQueue, otherQueue); // blocks myQueue until otherQueue’s ops have completed
```

alpaka in a Nutshell: Queue Data Structure

Setting up Accelerator, Device and Queue

```
// Choose types for dimensionality and indices
using Dim = dim::DimInt<1>;
using Idx = std::size_t;

// Choose the back-end
using Acc = acc::AccGpuHipRt<Dim, Idx>;

// Obtain first device in the HIP GPU list
auto myDev = pltf::getDevByIdx<Acc>(0u);

// Create non-blocking queue for chosen device
using Queue = queue::Queue<Acc, queue::NonBlocking>;
auto myQueue = Queue{myDev};

// Done! We can now enqueue device-side operations.
```

alpaka in a Nutshell: Task and Event

Tasks and Events

- Device-side operations (kernels, memory operations) are called Tasks
- Tasks on the same queue are executed in order (FIFO principle)

```
queue::enqueue(queueA, task1);
queue::enqueue(queueA, task2); // task2 starts after task1 has finished
```

- Order of tasks in different queues is unspecified

```
queue::enqueue(queueA, task1);
queue::enqueue(queueB, task2); // task2 starts before, after or in parallel to task1
```

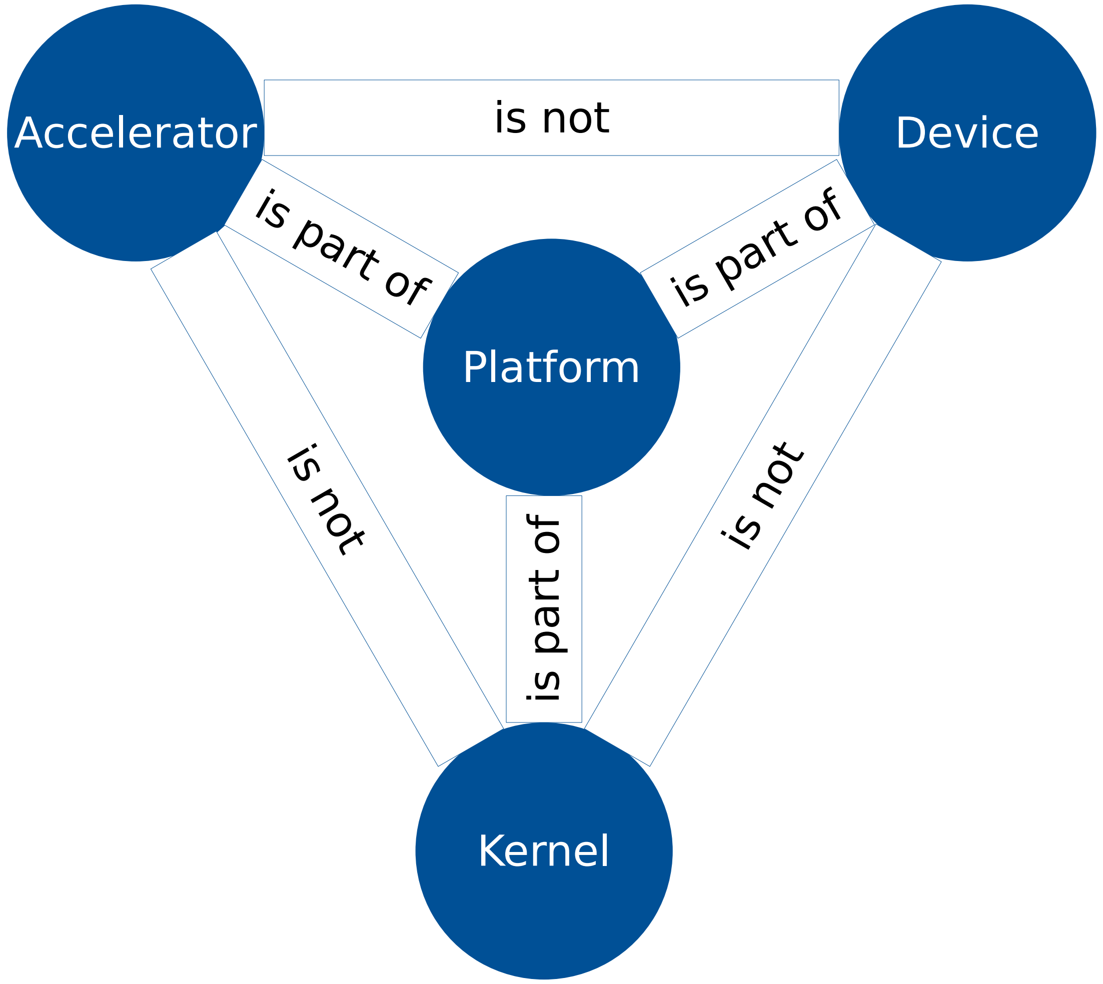
- For easier synchronization, alpaka Events can be inserted before, after or between Tasks:

```
auto myEvent = event::Event<Queue>(myDev);
queue::enqueue(queueA, myEvent);
wait::wait(queueB, myEvent); // queueB will only resume after queueA reached myEvent
```

alpaka in a Nutshell: The Platform

alpaka Platform

- Platform is meta-concept in alpaka
- Union of Accelerator, Device and Kernel functionality
  - Accelerator gives structure to the host side and functionality to the device side
  - Device gives functionality to the host side
  - Kernels are agnostic of Device details → Portable Kernels



alpaka in a Nutshell: The Platform

Changing the target platform

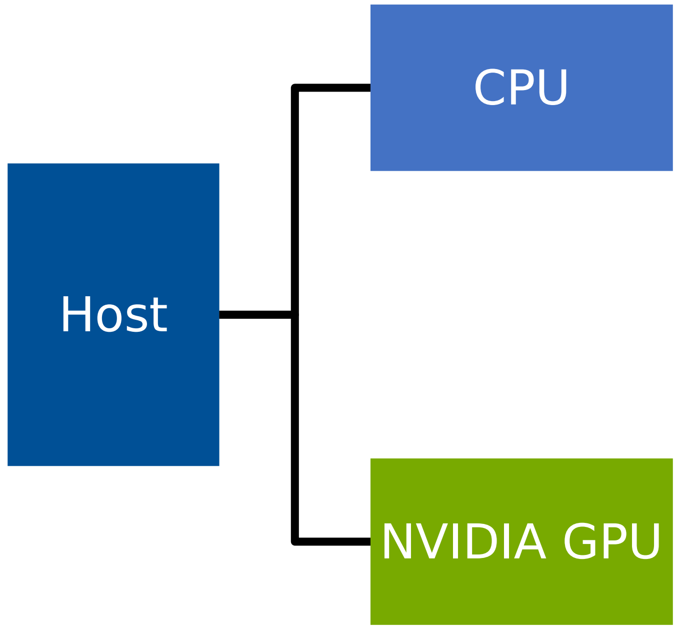
```cpp
using namespace alpaka;

using Dim = dim::DimInt<1u>;
using Idx = std::size_t;

/*** BEFORE ***/
using Acc = acc::AccCpuOmp2Blocks<Dim, Idx>;

/*** AFTER ***/
using Acc = acc::AccGpuHipRt<Dim, Idx>;

/* No change required - dependent types and variables are automatically changed */
auto myDev = pltf::getDevByIdx<Acc>(0u);

using Queue = queue::Queue<Acc, queue::NonBlocking>;
auto myQueue = Queue{myDev};
```

Summary of Alpaka Structures

- **Accelerator** provides abstract view of all capable physical devices
- **Device** represents a single physical device
- **Queue** enables communication between the host and a single Device
- **Platform** is a union of Accelerator, Device and Kernel
- **Task** is a device-side operation (e.g kernel, memory operation)
- Others: **Event, Buffer** (dynamic array), **Vector** (static array)
- **Question**: How is portability between back-ends achieved?

Programming Heterogeneous Systems

- Real-world scenario: Use all available compute power
- Also real-world scenario: Multiple different hardware types available
- Requirement: Usage of one back-end per hardware platform
- Requirement: Back-ends need to be interoperable



Programming Heterogeneous Systems

Why Alpaka – 4: Using multiple Platforms Synchronously

- Alpaka enables easy heterogeneous programming!
- Create one Accelerator per back-end
- Acquire at least one Device per Accelerator
- Create one Queue per Device

```
// Define Accelerators
using AccCpu = acc::AccCpuOmp2Blocks<Dim, Idx>;
using AccGpu = acc::AccGpuCudaRt<Dim, Idx>;

// Acquire Devices
auto devCpu = pltf::getDevByIdx<AccCpu>(0u);
auto devGpu = pltf::getDevByIdx<AccGpu>(0u);

// Create Queues
using QueueProperty = queue::NonBlocking;
using QueueCpu = queue::Queue<AccCpu, QueueProperty>;
using QueueGpu = queue::Queue<AccGpu, QueueProperty>;

auto queueCpu = QueueCpu{devCpu};
auto queueGpu = QueueGpu{devGpu};
```

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Programming Heterogeneous Systems

Communication by Buffers

- Buffers are defined and created per Device
- Buffers can be copied between different Devices / Queues
- Not restricted to a single platform!
- **Restriction**: CPU to GPU copies (and vice versa) require GPU queue

```
// Allocate buffers
auto bufCpu = mem::buf::alloc<float, Idx>(devCpu, extent);
auto bufGpu = mem::buf::alloc<float, Idx>(devGpu, extent);

/* Initialization … */

// Copy buffer from CPU to GPU - destination comes first
mem::view::copy(gpuQueue, bufGpu, bufCpu, extent);

// Execute GPU kernel
queue::enqueue(gpuQueue, someKernelTask);

// Copy results back to CPU and wait for completion
mem::view::copy(gpuQueue, bufCpu, bufGpu, extent);

// Wait for GPU, then execute CPU kernel
wait::wait(cpuQueue, gpuQueue);
queue::enqueue(cpuQueue, anotherKernelTask);
```

alpaka In a nutshell

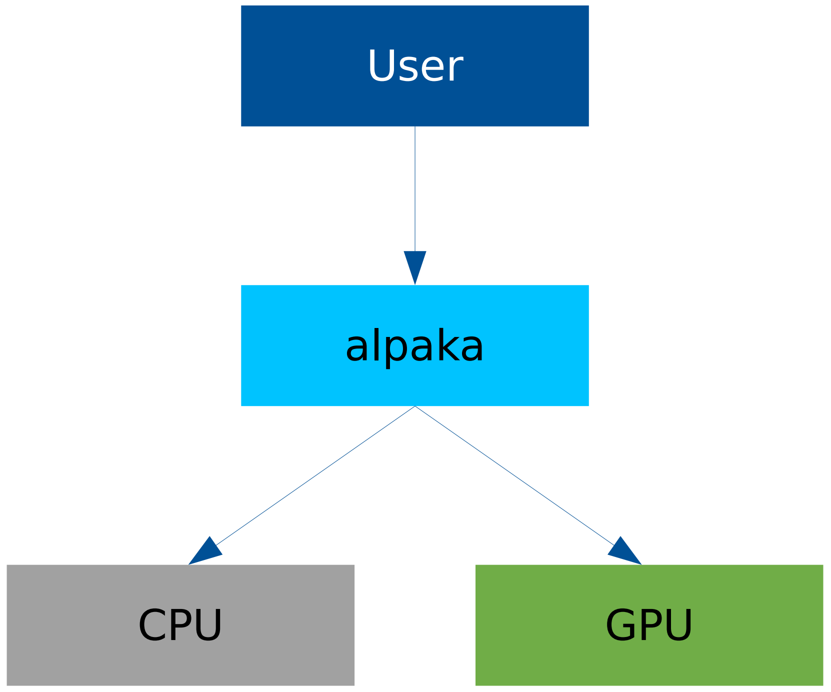
As a summary

Without alpaka

- Multiple hardware types commonly used (CPUs, GPUs, …)
- Increasingly heterogeneous hardware configurations available
- Platforms not inter-operable → parallel programs not easily portable

alpaka: one API to rule them all

- **Abstraction** (not hiding!) of the underlying hardware & software platforms
  - AMD, Nvidia, Intel GPUs, CPU
- **Easy change of the backend**
  - Code needs only minor adjustments to support different accelerators
- **Easy indexing of threads in kernels**
- **Easy setup of the type of parallelism** (Block sizes in grid, Thread sizes in block…)
- **Heterogenous Programming**: Using different backends in a synchronized manner



alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Thank you! If you use alpaka for your research, please cite one of the following publications:

Matthes A., Widera R., Zenker E., Worpitz B., Huebl A., Bussmann M. (2017): Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the Alpaka Library. In: Kunkel J., Yokota R., Taufer M., Shalf J. (eds) High Performance Computing. ISC High Performance 2017. Lecture Notes in Computer Science, vol 10524. Springer, Cham, DOI: **10.1007/978-3-319-67630-2_36**.

E. Zenker et al., “Alpaka – An Abstraction Library for Parallel Kernel Acceleration”, 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, 2016, pp. 631 – 640, DOI: **10.1109/IPDPSW.2016.50**.

Worpitz, B. (2015, September 28). Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures. Zenodo. DOI: **10.5281/zenodo.49768**.

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science