Efficient Scientific Computing School – 12th Edition

Performance Portability With *alpaka*

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF
TECHNISCHE UNIVERSITÄT DRESDEN
CBG Max Planck Institute of Molecular Cell Biology and Genetics
UFZ HELMHOLTZ Centre for Environmental Research
Uniwersytet Wrocławski

SPONSORED BY THE
Federal Ministry of Education and Research
STAATSMINISTERIUM FÜR WISSENSCHAFT KULTUR UND TOURISMUS
Freistaat SACHSEN

Life After CUDA

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Previously on ESC21 …

- Parallel programming models:
  - Intel TBB
  - `std::thread`
  - NVIDIA CUDA
- Many more available:
  - AMD HIP
  - OpenCL & SYCL / Intel oneAPI
  - OpenMP
  - OpenACC
  - Boost.Fiber
  - …
- Challenge: How to keep programs portable?

Introduction to alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

Introduction to alpaka

Programming with alpaka

- C++ only!
- Header-only library: No additional runtime dependency introduced
- Modern library: alpaka is written entirely in C++14, transitioning to C++17 soon
- Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MS Visual Studio)
- Portable across operating systems: Linux, macOS, Windows are supported

Introduction to alpaka

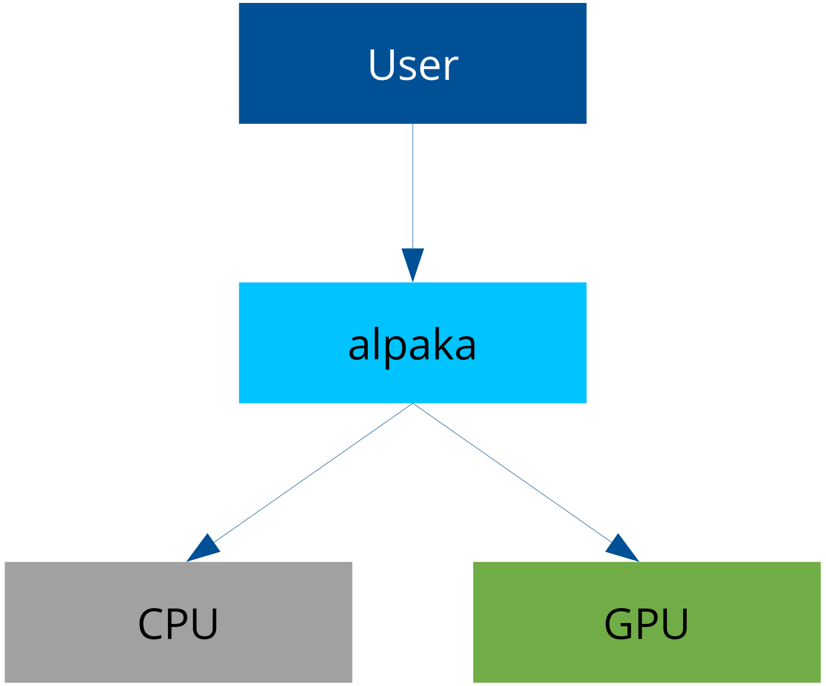
alpaka's purpose

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

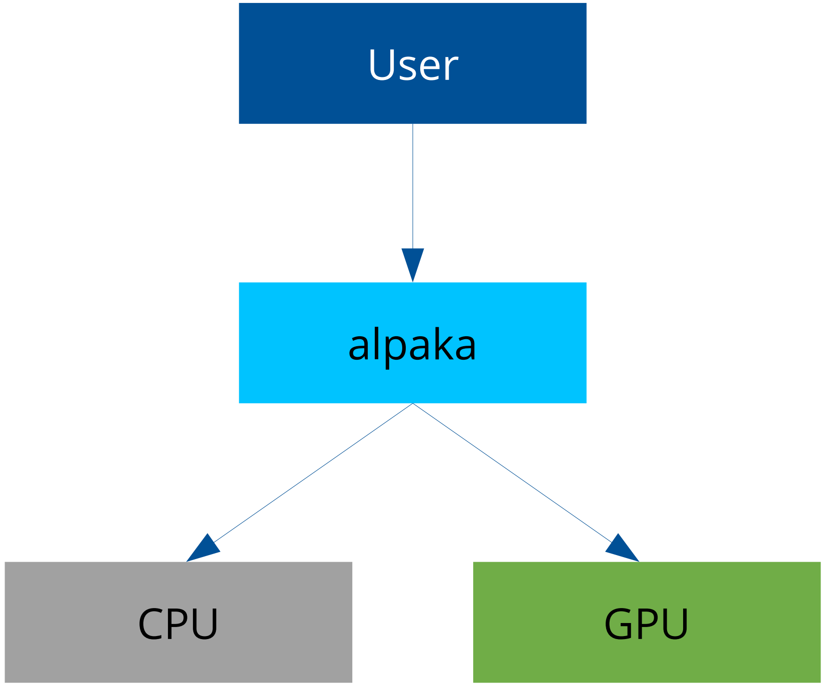
- Abstraction (not hiding!) of the underlying hardware & software platforms
- Code needs only minor adjustments to support different accelerators



Portable Heterogeneous Parallel Programming

alpaka enables portability!

- Idea: Write algorithms once…
  - … independently of target architecture
  - … independently of available programming models
- Decision on target platform made during compilation
  - Choosing another platform just requires another compilation pass
- alpaka defines an abstract programming model
- alpaka utilizes C++14 to support many architectures
  - CUDA, HIP, OpenMP, TBB, …



Portable Heterogeneous Parallel Programming

alpaka enables full utilization of heterogeneous systems!

- Algorithms are generally independent of chosen target architecture

```
auto const taskCpu = alpaka::createTaskKernel<AccCpu>(workDivCpu, kernel, …);
auto const taskGpu = alpaka::createTaskKernel<AccGpu>(workDivGpu, kernel, …);
```

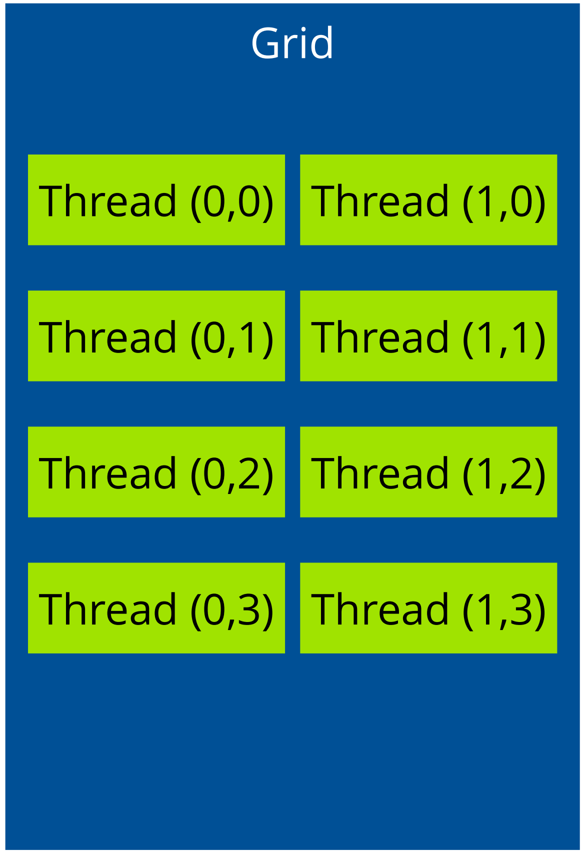
- Optimization for specific architecture is still possible

```
// general case
template <typename TAcc>
void computationallyIntensiveFunction(TAcc const & acc) { … };

// specialization for AccGpu
template <>
void computationallyIntensiveFunction<AccGpu>(AccGpu const & acc) { … };
```

Portable Heterogeneous Parallel Programming

How parallelism is achieved, Part I: The grid, a digital frontier

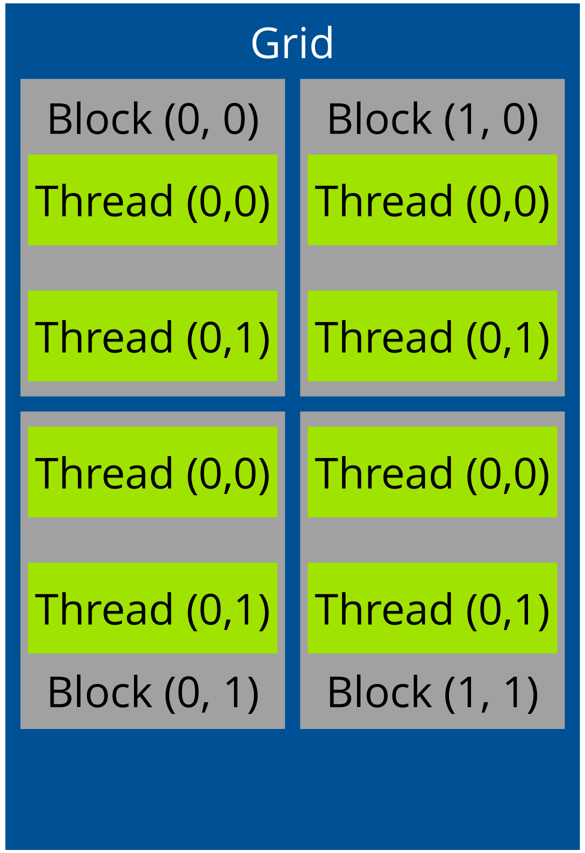
- alpaka is ideal for data-parallel algorithms
  → execute the same algorithm on different data elements
- alpaka **kernel**: sequence of commands forming the algorithm on a per-element level
- alpaka **thread:** execution of a kernel for a single (execution) element
- threads are executed in parallel and are independent of each other
- alpaka **grid**: n-dimensional grid of all **threads** executing a specific kernel
  - each thread is assigned a unique index on the grid
  - threads on the grid are able to communicate through high-latency **global memory**



Portable Heterogeneous Parallel Programming

How parallelism is achieved, Part II: Blocks on the grid

- Grids are divided into independent **blocks** of equal size
- Each thread is assigned to exactly one block
- Each thread is assigned an unique index on the block
- All threads inside a block are executed in parallel
- All threads inside a single block can be **synchronized**
  → no synchronization on the grid level!
- All threads inside a block can communicate through low-latency **shared memory**



Portable Heterogeneous Parallel Programming

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Summary

- alpaka is ideal for data-parallel algorithms
- Algorithms are written per data element (**kernel**)
- data parallelism achieved through a hierarchy of independent **threads** and **blocks** on a **grid**
- All threads can communicate through high-latency **global memory**
- Threads inside a block can be **synchronized**
- Threads inside a block can communicate through low-latency **shared memory**

Download & Installation

Download

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

How to download alpaka

- Install `git` for your operating system:
  - Linux: `sudo dnf install git` (RPM) or `sudo apt install git` (DEB)
  - macOS: Enter `git --version` in your terminal, you will be asked if you want to install `git`
  - Windows: Download the installer from **https://git-scm.com/download/win**
- Open the terminal (Linux / macOS) or PowerShell (Windows)
- Navigate to a directory of your choice: `cd /path/to/some/directory`
- Download alpaka: `git clone -b 0.7.0 https://github.com/alpaka-group/alpaka.git`

Dependencies

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Install alpaka's dependencies

- alpaka only requires Boost and a modern C++ compiler (`g++`, `clang++`, Visual C++, …)
  - Linux:
    - `sudo dnf install boost-devel` (RPM)
    - `sudo apt install libboost-all-dev` (DEB)
  - macOS:
    - `brew install boost` (Using Homebrew, **https://brew.sh**)
    - `sudo port install boost` (Using MacPorts, **https://macports.org**)
  - Windows: `vcpkg install boost` (Using vcpkg, **https://github.com/microsoft/vcpkg**)
- Depending on your target platform you may need additional packages
  - NVIDIA GPUs: CUDA Toolkit (**https://developer.nvidia.com/cuda-toolkit**)
  - AMD GPUs: ROCm / HIP (**https://rocmdocs.amd.com/en/latest/index.html**)

Installation

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

(Optional) Install alpaka's headers

- alpaka is already ready to use!
- Create an installation directory for the headers:
  ```
  mkdir /some/install/dir/
  ```
- Copy the alpaka headers to the new directory:
  ```
  cp -r alpaka/include /some/install/dir
  ```

Test Your Installation

- Create a small program that includes the main alpaka header:

```cpp
#include <alpaka/alpaka.hpp>

#include <cstdlib>

int main()
{
    return EXIT_SUCCESS;
}
```

- Compile:

```
$ nvcc -std=c++14 -I/some/install/dir/include tutorial.cpp
$ ./a.out
```

- Add the following compiler flag to silence the warnings:

```
-Xcudafe=--diag_suppress=esa_on_defaulted_function_ignored
```

Exercise 0: Set up alpaka on your system!

Before we proceed...

Lecture Notes

- Find the cheatsheet: **https://alpaka.readthedocs.io/en/0.7.0/basic/cheatsheet.html**
- Assume `using namespace alpaka;` everywhere!

Calculating AXPY

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow a \cdot \vec{x} + \vec{y}$$

```
struct AxpyKernel
{
    template <typename TAcc>
    ALPAKA_FN_ACC void operator()(TAcc const& acc,
      std::size_t numElements, int a, int const* X, int* Y) const
    {
        auto gridThreadIdx = getIdx<Grid, Threads>(acc)[0u];
        auto threadElems = getWorkDiv<Thread, Elems>(acc)[0u];
        auto first = gridThreadIdx * threadElems;

        if(first < numElements)
        {
            auto last = first + threadElems;
            for(auto i = first; i < last; ++i)
                Y[i] = a * X[i] + Y[i];
        }
    }
};
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow a \cdot \vec{x} + \vec{y}$$

```
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;

auto const host = getDevByIdx<DevCpu>(0u);
auto const dev = getDevByIdx<Acc>(0u);
using myQueue = Queue<Acc, property::Blocking>;
auto queue = myQueue{dev};

auto const ext = Vec<Dim, Idx>{1024};
auto hostBufY = allocBuf<int, Idx>(host, ext);
/* Initialize … */
auto devBufY = allocBuf<int, Idx>(dev, ext);

memcpy(queue, devBufY, hostBufY, ext);

auto workDiv = getValidWorkDiv<Acc>(dev, ext, Idx{1u});
auto taskKernel = createTaskKernel<Acc>(
                    workDiv, AxpyKernel{}, /* params … */);
enqueue(queue, taskKernel);

memcpy(queue, hostBufY, devBufY, ext);
```

alpaka's Programming Model

Handling Parallelism

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

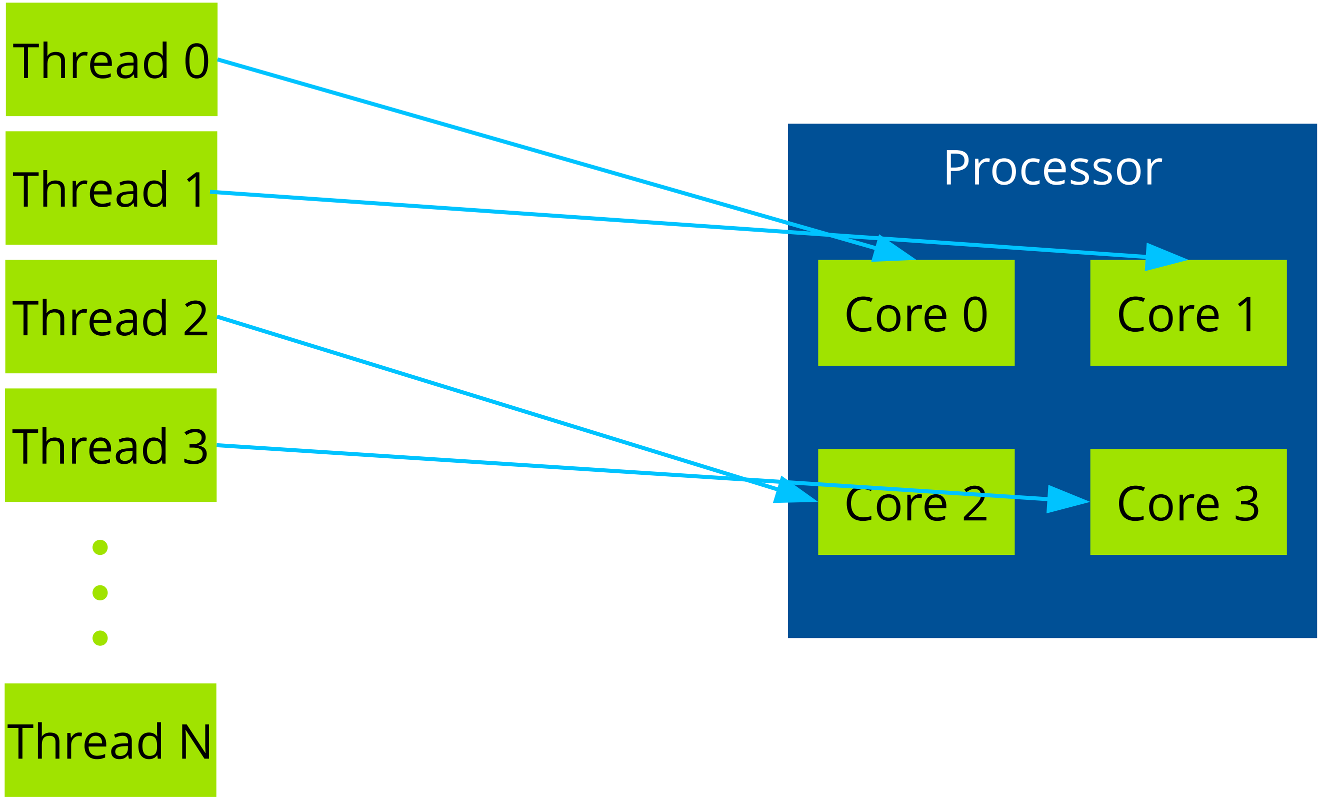
Threads and cores

- alpaka Threads are different from pthreads, `std::threads`, OpenMP threads, CUDA threads, etc.
- alpaka Thread: execution of command sequence
- Command sequence: algorithm performed on single data element (Kernel)
- Cores are physical execution units
- Cores are capable of executing alpaka Threads
- Example: AMD Threadripper 3990X with 64 CPU cores
- Example: NVIDIA Tesla V100 with 5,120 CUDA cores

Handling Parallelism

Mapping Threads to cores

- alpaka Threads are mapped to hardware cores
- While running, one Thread is executed by exactly one core
- Threads may run on other cores after rescheduling
- Usually many more Threads than cores (*oversubscription*)
- Waiting Threads make room for ready Threads



Handling Parallelism

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

alpaka Devices

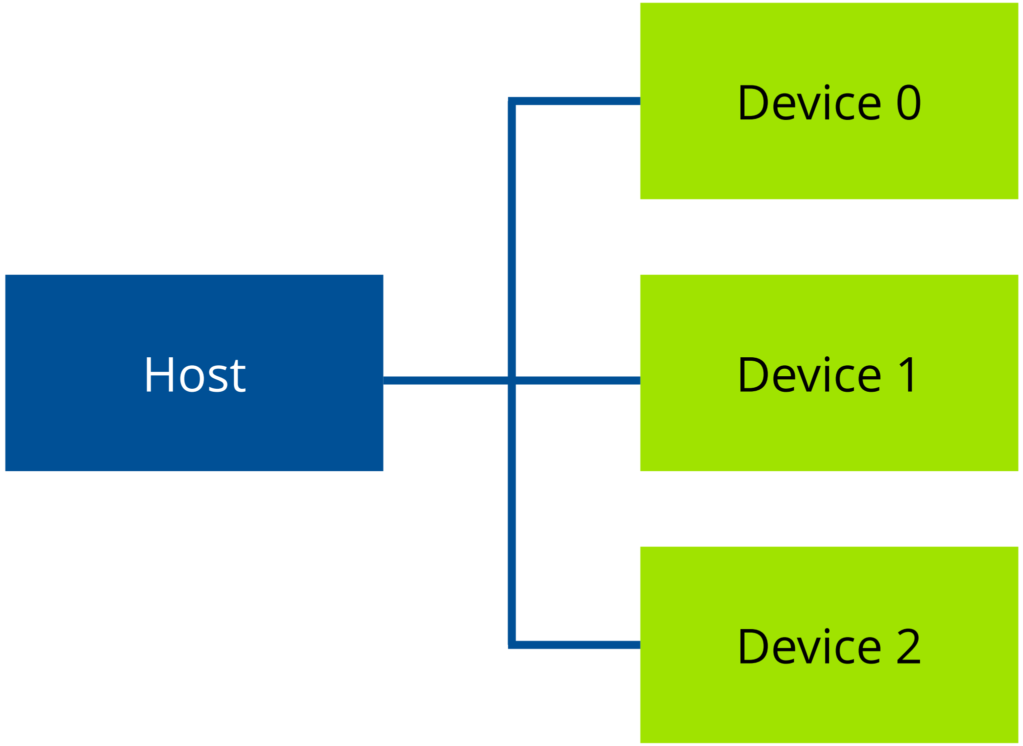
- A set of cores is called a Device
- A single core can only belong to exactly one Device (N:1 mapping)
- All cores on the Device have access to global memory
- alpaka Devices correspond to physical devices
- Example: AMD Threadripper 3990X with 64 CPU cores is a Device with 128 cores (simultaneous multithreading!)
- Example: NVIDIA Tesla V100 with 5,120 CUDA cores is a Device with 5,120 cores

Handling Parallelism

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Host and Device

- An alpaka Host controls the overall program flow
- An alpaka Device executes Kernels
- All Devices are attached to a single Host
- It is impossible to have more than one Host per process



Kernels

What is a 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`

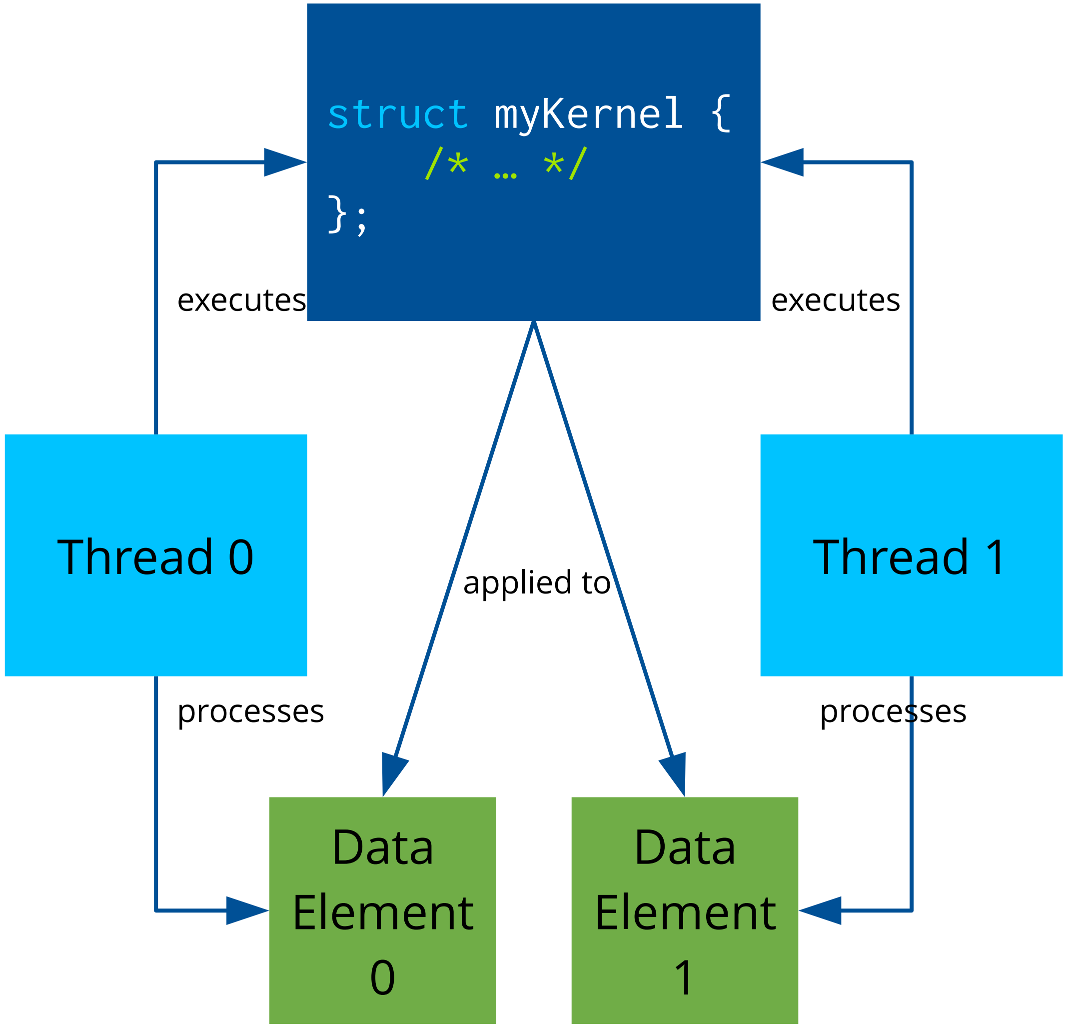
```
struct HelloWorldKernel
{
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc) const
    {
        uint32_t threadIdx = getIdx<Grid, Threads>(acc)[0];

        printf("Hello, World from alpaka thread %u!\n", threadIdx);
    }
};
```

Kernels

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



Kernels

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Scheduling

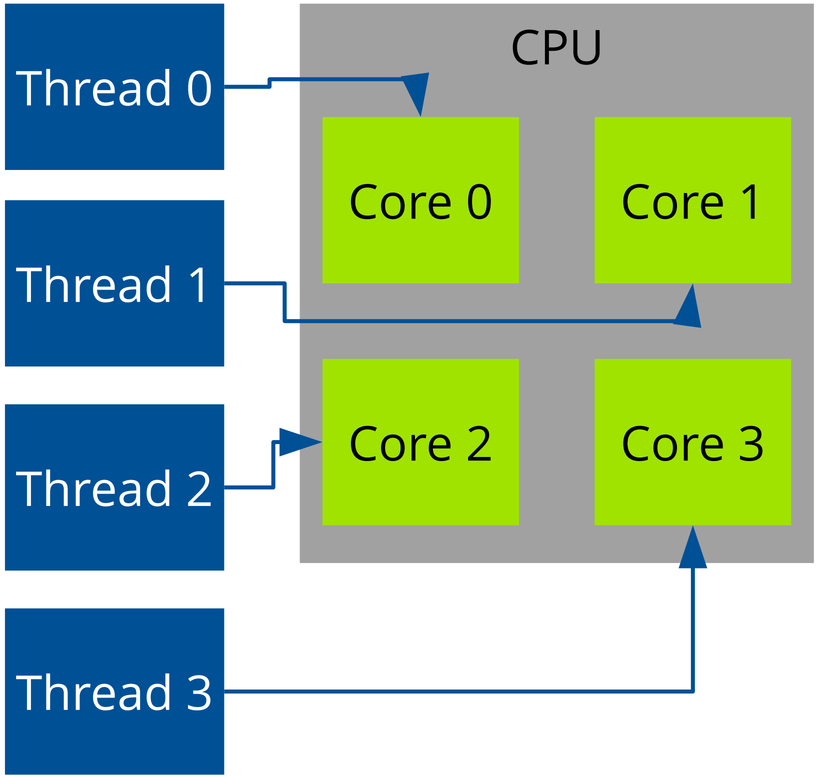
- Threads are mapped to cores
- Many more Threads than cores → Thread scheduling required
- **Thread order is unspecified!**

  → Programmer cannot control the order of element processing
- Hardware specifics need to be taken into account

Kernels

Example: Thread mapping on CPUs

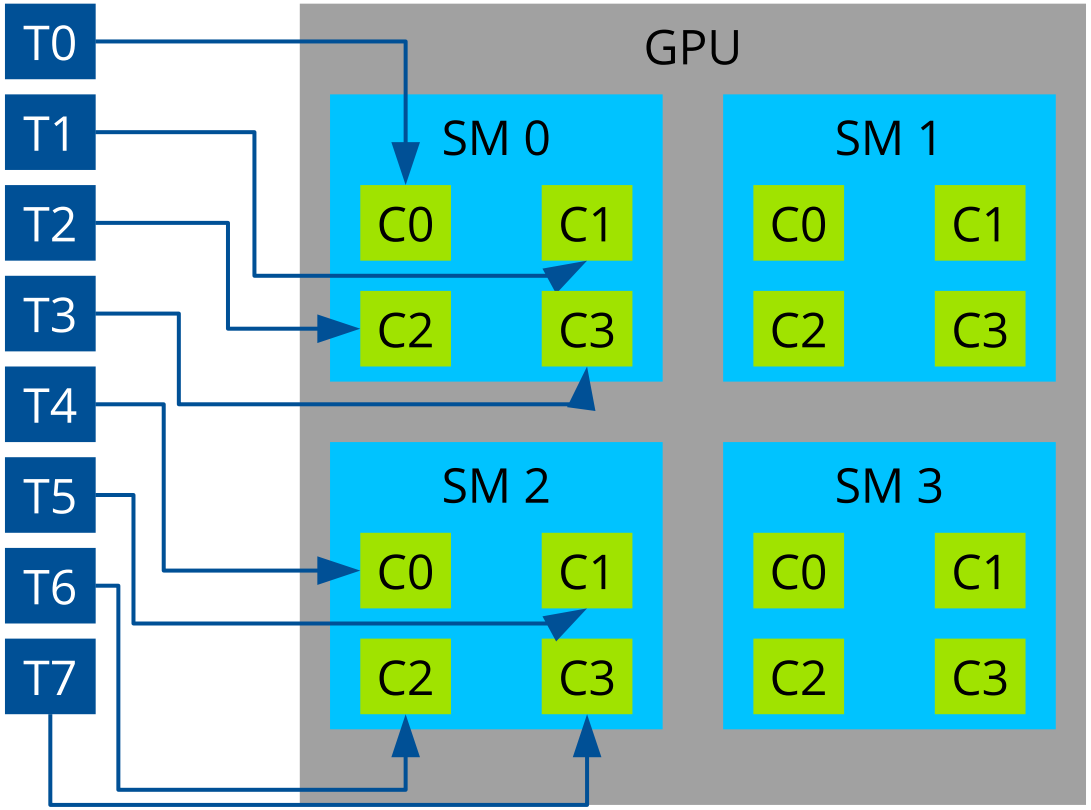
- CPU consists of multiple cores
  - Because of simultaneous multithreading there can be more logical than physical cores!
- alpaka Threads are executed by CPU cores



Kernels

Example: Thread mapping on GPUs

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



The Problem Size

Problem size and hardware capabilities

- The programmer's questions:
  - How large is the problem? (= How many data elements need processing?)
  - Which capabilities are offered by the hardware? (= How many cores are available?)
- The programmer's challenge:
  - Problem size and number of cores completely disjoint
  - How to distribute the former amongst the latter?

The Problem Size

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

How to choose the number of alpaka Threads

- The two important factors:
  - Problem size → number of data elements
  - Hardware capabilities → number of cores
- Rule of thumb: One Thread per data element
  - Not always ideal (depending on algorithm)
  - Chance for optimisation

The Problem Size

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Choosing the number of Threads

- (Usually) you have more Threads than cores
- In alpaka, the overall number of Threads is `blocksPerGrid * threadsPerBlock`
  - We will introduce Thread Blocks later!

```
using Idx = std::uint32_t;

Idx blocksPerGrid = 8;
Idx threadsPerBlock = 1;
```

The Problem Size

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Beware!

- Don’t run too many Threads in parallel!
  - An exact definition of “too many” depends on your hardware.
- Some hardware resources are always shared between Threads
- Having too many Threads accessing shared resources results in bottlenecks
  - Can seriously impact your program’s performance
  - Chance for optimisation

The Problem Size

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Example: I/O buffer

- All Threads call `printf`
- The access to the output buffer needs to be serialized
- More Threads
  → more serialization
  → worse performance

```
template <typename Acc>
ALPAKA_FN_ACC void operator()(Acc const & acc) const
{
    auto threadIdx = getIdx<Grid, Threads>(acc)[0];
    printf("Hello, World from alpaka thread %u!\n", threadIdx);
}
```

Thread Hierarchy

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

The "magic" Thread index

```
template <typename Acc>
ALPAKA_FN_ACC void operator()(Acc const & acc) const
{
    auto threadIdx = getIdx<Grid, Threads>(acc)[0];
    printf("Hello, World from alpaka thread %u!\n", threadIdx);
}
```

Thread Hierarchy

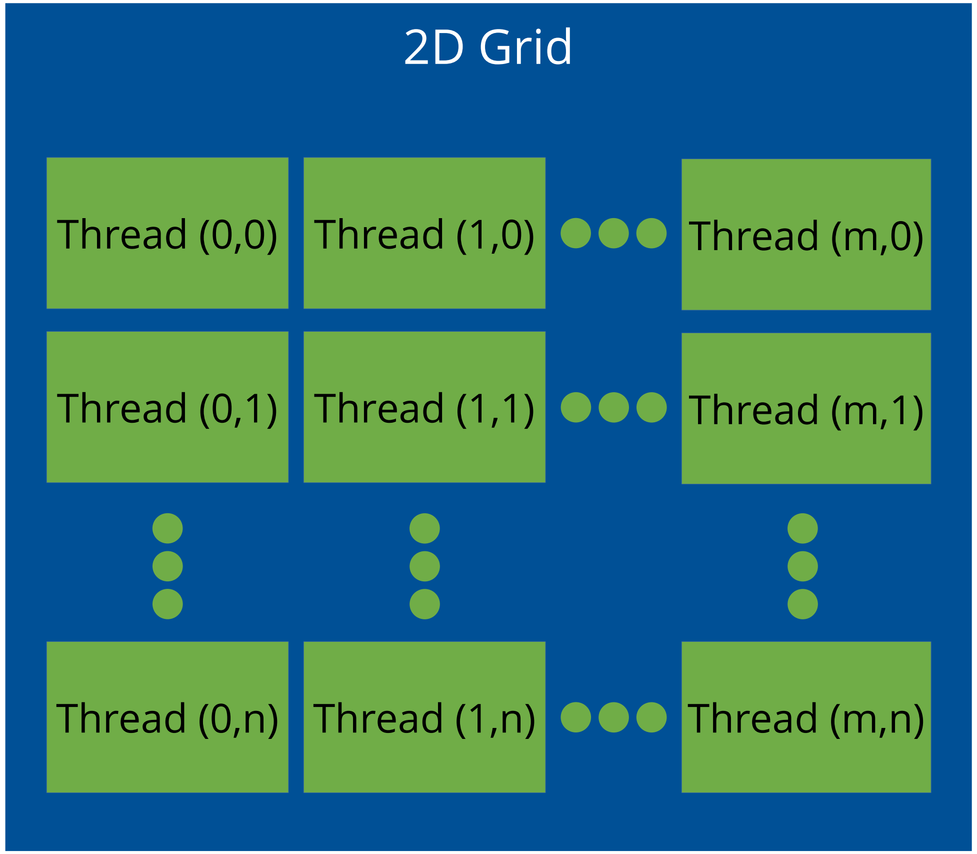
Understanding the index

- Understanding alpaka's Thread indices is the key to understanding alpaka!
- After this section, you will understand:
  - How to navigate the grid
  - How to form Thread Blocks (and why)
  - The relations between Threads, Blocks and the Grid
  - How to compute Thread indices yourself

Thread Hierarchy

Threads and the Grid

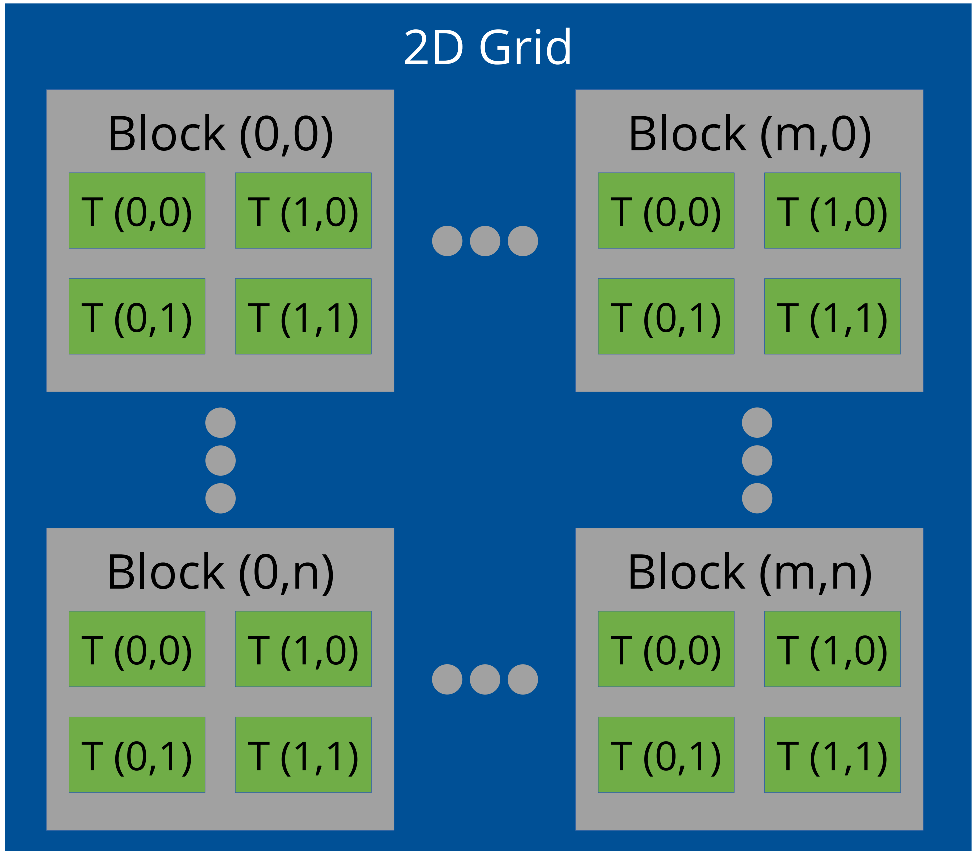
- A Grid consists of all Threads executing the same kernel
  → One Grid per Kernel execution
- Threads are distributed along one, two or three dimensions
- Each Thread on the Grid is identified by its unique index (`gridThreadIdx`)
- All Threads have access to (large but high-latency) global memory



Thread Hierarchy

Thread Blocks

- Threads can be grouped into Thread Blocks
- All Blocks on the same Grid have the same size
- Each Block on the Grid is identified by its unique index (`gridBlockIdx`)
- Each Thread inside a Block is identified by its Block-local unique index (`blockThreadIdx`)
- Threads inside a Block have access to (small but low-latency) shared memory
- Threads inside a Block can be synchronized



Thread Hierarchy

Obtaining the indices

- alpaka provides several API functions for obtaining indices:
  - Index of Thread on the Grid: `getIdx<Grid, Threads>(acc)[dim];`
  - Index of Thread on a Block: `getIdx<Block, Threads>(acc)[dim];`
  - Index of Block on the Grid: `getIdx<Grid, Blocks>(acc)[dim];`
- You can also obtain the extents of the Grid or the Blocks:
  - Number of Threads on the Grid: `getWorkDiv<Grid, Threads>(acc)[dim];`
  - Number of Threads on a Block: `getWorkDiv<Block, Threads>(acc)[dim];`
  - Number of Blocks on the Grid: `getWorkDiv<Grid, Blocks>(acc)[dim];`
- Exercise: compute the index of a Thread on the Grid yourself using a combination of the remaining indices and extents!

2D Work Division

From 1D to 2D

- *n*-dimensional grids work in a similar way to 1D grids
  - `getIdx<Grid, Threads>(acc)` returns a vector containing *n* indices
  - `getIdx<Grid, Threads>(acc)[dim]` returns an integer
- **Beware**: In a 2D grid, *y* is dimension zero and *x* is dimension one
  - `getIdx<Grid, Threads>(acc)` returns a vector containing 2 indices: the *y*-index at position `0` and the *x*-index at position `1`
  - `getIdx<Grid, Threads>(acc)[0]` returns the *y*-index

Computing π

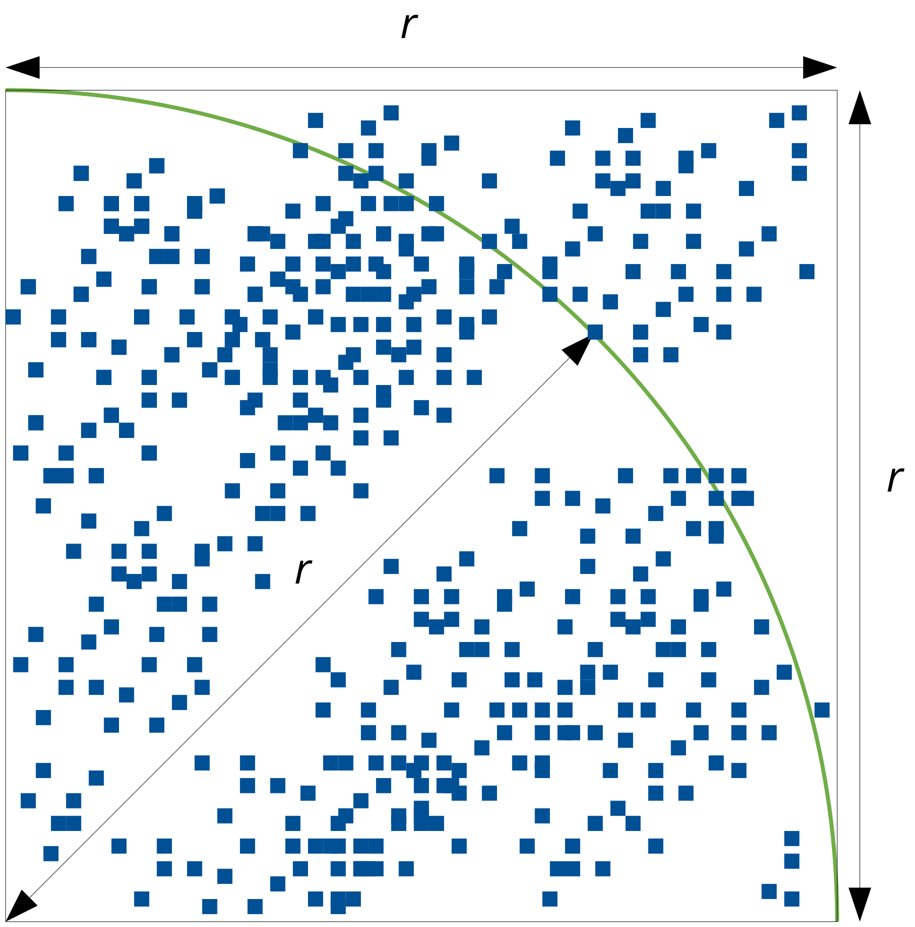
Computing π

- Focus of the next four lessons
- Good example for Thread parallelism
- Introduces parameter passing and memory management
- Initial algorithm: Find points in a circle

Computing π – Part I

Points in a circle

- Task: Given a circle quarter with the radius *r* and a set of *n* randomly scattered points, find all points inside the circle quarter
- Approach:
  - Create a Grid with *n* Threads
  - Each Thread evaluates a single point



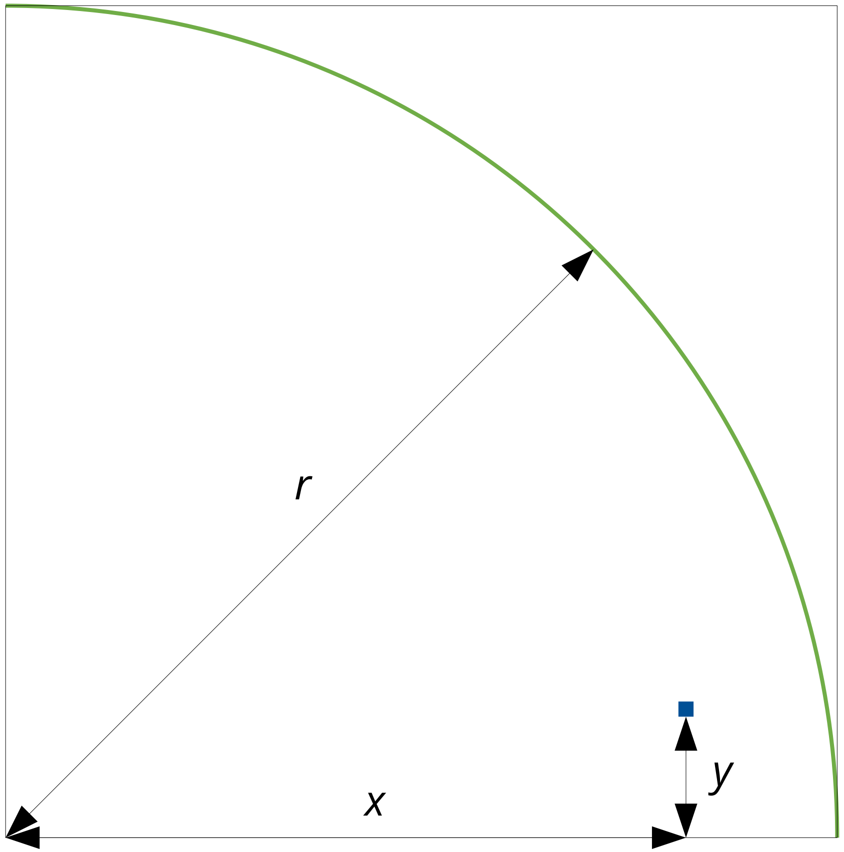
Computing π – Part I

Algorithm

- Using Pythagoras' theorem, the distance *d* from a point to the origin can be calculated:

  $$d = \sqrt{x^2 + y^2}$$

- If $d \leq r$, return `true`, otherwise `false`



Computing π – Part I

Kernel requirements

- For the computation we need:
  - The point coordinates:
    ```
    struct Points {
        float* x;
        float* y;
        bool* inside;
    };
    ```
  - The radius: `float r;`
- How do we pass these to the kernel?

Computing π – Part I

Passing parameters

- alpaka kernels accept three different parameter types:
  - The accelerator: `Acc const & acc` (required)
  - Pointers to memory buffers of any data type: `float* bufferA`, `MyDataType* bufferB`
  - Scalar values of trivially copyable types: `float scalar`, `struct Composed { int a; float b; }`
- Signature of the `PixelFinderKernel`'s `operator()`:

```
template <typename Acc>
ALPAKA_FN_ACC void operator()(Acc const & acc, // required
                              Points points,   // this struct contains memory buffers
                              float r          // this is a scalar
) const
```

Computing π – Part II

Grid dimensionality

- No spatial relationship between points
- Points can be evaluated independently
- This makes a multi-dimensional grid unnecessary

```
struct PixelFinderKernel
{
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc, Points points, float r) const {

        uint32_t gridThreadIdx = getIdx<Grid, Threads>(acc)[0];
        /* … */
```

Computing π – Part II

Accessing memory

- Iterating over a buffer works differently in alpaka
- `for` loop: One thread accesses elements sequentially
- Thread index: Threads access elements in parallel
- If required, you can mix both approaches!

```
// Using a for loop for buffer access
for(std::size_t i = 0; i < n; ++i)
{
    float x = points.x[i];
    float y = points.y[i];
}
```

```
// Using the thread index for buffer access
float x = points.x[gridThreadIdx];
float y = points.y[gridThreadIdx];
```

Computing π – Part II

Computing the distance

- Use Pythagoras' theorem for computing the distance
- Use `sqrt()` for computing the square root
  - Requires the `acc` parameter!

```
        /* … */
        float d = sqrt(acc, x * x + y * y);

        bool isInside = (d <= r);

        points.inside[gridThreadIdx] = isInside;
    }
};
```

Computing π – Part II

The complete Kernel

```
struct PixelFinderKernel
{
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc, Points points, float r) const {

        uint32_t gridThreadIdx = getIdx<Grid, Threads>(acc)[0];

        float x = points.x[gridThreadIdx];
        float y = points.y[gridThreadIdx];
        float d = sqrt(acc, x * x + y * y);

        bool isInside = (d <= r);

        points.inside[gridThreadIdx] = isInside;
    }
};
```

Computing π – Part III

Kernel requirements

- alpaka kernels accept pointers to Device memory
- Challenge: Host and Device don’t always share memory
- Memory buffers need to be allocated on both the Host and the Device
- Memory needs to be transferred from the Host to the Device and vice versa
- In case of CPU Devices there is optimisation potential in avoiding unnecessary copies!

Computing π – Part III

Allocating memory on the Host

- Memory can be allocated using `allocBuf()`

```
using Host = /* … */;                                   // not important now
using BufHost = Buf<Host, float, Dim, Idx>;             // Host buffer type
using MyVec = Vec<Dim, Idx>;                            // Vector type

auto const devHost = getDevByIdx<Host>(0u);             // create host device
Vec const extents(n);                                   // create extents
BufHost hostBuffer = allocBuf<float, Idx>(devHost, extents);
```

- Pre-allocated memory can be used with alpaka:

```
std::vector<float> plainBuffer(n);
using ViewHost = ViewPlainPtr<Host, float, Dim, Idx>;
ViewHost hostViewPlainPtr(plainBuffer.data(), devHost, Vec(plainBuffer.size());
```

Computing π – Part III

Allocating memory on the Device

- Allocating memory on the Device works the same way!
- Memory can be allocated using `allocBuf()`

```
using Acc = /* … */;                                    // not important now
using BufAcc = Buf<Acc, float, Dim, std::size_t>;       // Accelerator buffer type

auto const devAcc = getDevByIdx<Acc>(0u);               // create accelerator device

BufAcc accBuffer = allocBuf<float, std::size_t>(devAcc, extents);
```

Computing π – Part III

Memory transfers

- After initializing the Host buffer (`for` loop, `<algorithm>`, `memset`, …) memory can be transferred
- In alpaka all memory operations are explicit
- Use `memcpy()` to initiate transfers:

```
memcpy(devQueue,            // queue (explained later)
       devBuffer,           // copy target
       hostBuffer,          // copy source
       extents);            // number of elements
```

```
memcpy(devQueue,
       devBuffer,
       hostViewPlainPtr,    // for pre-allocated memory
       extents);
```

Computing π – Part IV

Approach

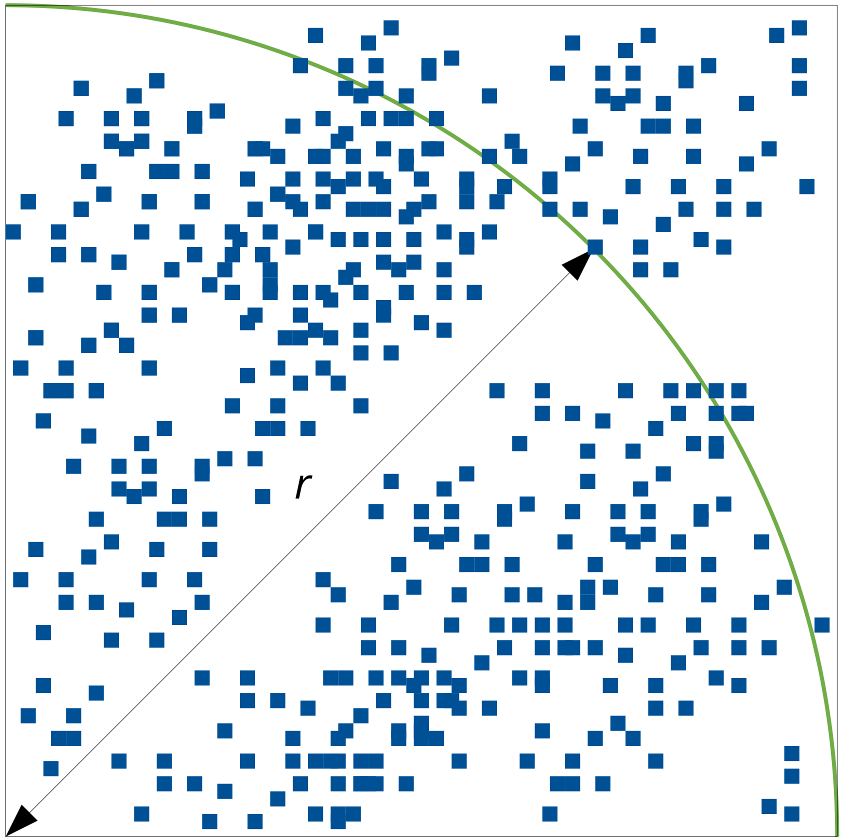
- We will use the formula for the area of a circle quarter:

  $$A = \frac{\pi \cdot r^2}{4}$$

- The number of points inside the circle (*P*) can be used to approximate *A:*

  $$\frac{P}{n} \approx \frac{A}{r^2} = \frac{\pi}{4} \rightarrow \pi \approx \frac{4P}{n}$$

- The `PixelFinderKernel` does the counting on the Device, integration is done by the Host.



Lesson 26: Computing π – Part IV

Kernel execution and memory transfer

- We will measure the execution time:

```
auto start = std::chrono::steady_clock::now();
```

- Execute the kernel using `enqueue()`:

```
PixelFinderKernel pixelFinderKernel;
auto taskRunKernel = createTaskKernel<Acc>(workDiv, pixelFinderKernel, pointsAcc, r);
enqueue(queue, taskRunKernel);
```

- Copy back the results and synchronize:

```
memcpy(devQueue, insideBufferHost, insideBufferAcc, extents);
wait(devQueue);
```

Computing π – Part IV

Integration

- First, determine *P*:

```
std::uint64_t P = 0;
for(std::size_t i = 0; i < n; ++i)
{
    if(pointsHost.inside[i])
        ++P;
}
```

- Then, divide by the radius to approximate π:

```
float pi = (4.f * P) / n;
```

- Measure the execution time:

```
auto end = std::chrono::steady_clock::now();
```

Computing π – Part IV

Aftermath

- Print out π and execution time:

```
std::chrono::duration<double, std::milli> duration = end - start;
std::cout << "Computed pi is " << pi << "\n";
std::cout << "Execution time: " << duration.count() << "ms" << std::endl;
```

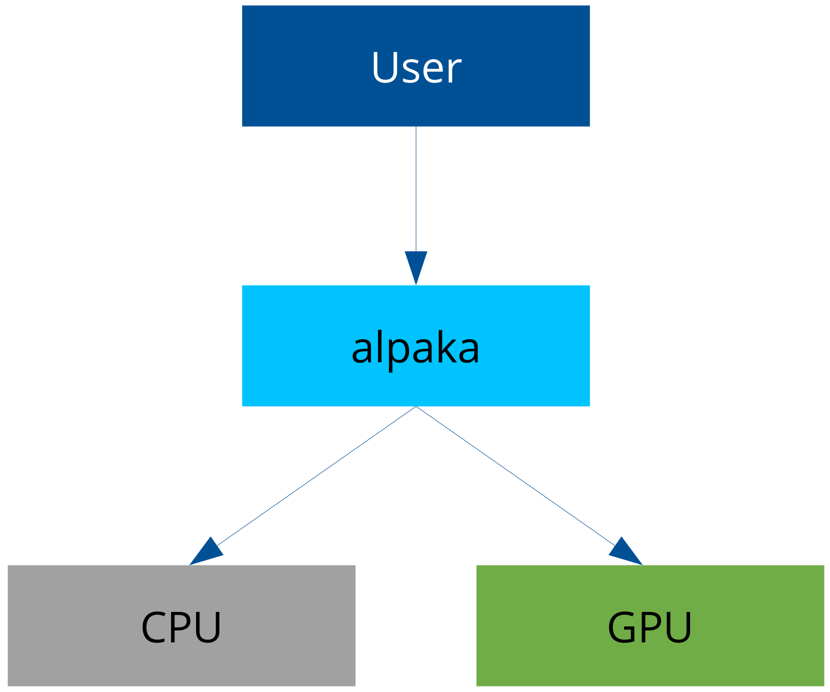
- Homework #1: Play around with *n*. How does this affect the precision of π and the execution time?
- Homework #2: Implement the kernel in a more generic way, so that it works for any number of threads, blocks and grids.
  - The workload has to be distributed between all threads in the grid.
  - It requires to have a loop over points inside the kernel.

Changing the Back-end

Moving from CPU to GPU

alpaka allows for easy …

- … exchange of the accelerator
- … porting of programs across accelerators
- … experimentation with different devices
- … mixing of accelerator types



Changing the Back-end

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Switching the Accelerator

- alpaka provides a number of pre-defined back-ends (called *Accelerators)*
- 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
  - `AccCpuOmp5` based on OpenMP 5.x
  - `AccCpuTbbBlocks` based on TBB
  - `AccCpuThreads` based on `std::thread`

```
// Example: CPU accelerator
using Acc = AccCpuOmp2Blocks<Dim, Idx>;

// Example: CUDA GPU accelerator
using Acc = AccGpuCudaRt<Dim, Idx>;

// Example: HIP GPU accelerator
using Acc = AccGpuHipRt<Dim, Idx>;
```

Changing the Back-end

Changing the work division

- GPUs have many more cores than CPUs
  → More parallel threads possible
- GPUs have several multiprocessors
- Each multiprocessor can execute multiple threads
- Threads are grouped into blocks
- Blocks are scheduled to run on multiprocessors

```
// CPU work division (example)
Idx blocksPerGrid     = 8;
Idx threadsPerBlock   = 1;
Idx elementsPerThread = 1;

// GPU work division (example)
Idx blocksPerGrid     = 64;
Idx threadsPerBlock   = 512;
Idx elementsPerThread = 1;
```

Changing the Back-end

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

GPU performance hints

- Avoid divergent `if`-`else`-blocks
  - GPU threads are organized into groups (NVIDIA: *warp*, AMD: *wavefront*)
  - Groups are executed in lock step
    → If there is divergence, all threads execute the `if` block first and the `else` block next
- GPU threads are much more lightweight than CPU threads
  - Context switch is much cheaper on GPUs
  - Spawn many more threads than you have GPU cores
    → Hide memory latency behind computation

The Accelerator Concept

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Introduction

- alpaka's Accelerator concept is an important tool
- Accelerator hides hardware specifics behind alpaka's abstract API
- Chosen by programmer:

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

- Used on both Host and Device
- Inside Kernel: contains thread state, provides access to alpaka's device-side API
- On Host: Meta-parameter for choosing correct physical device and dependent types

The Accelerator Concept

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Accelerators and devices

- 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 = AccCpuOmp2Blocks<Dim, Idx>;

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

```
/* After the code change */
using Acc = AccGpuHipRt<Dim, Idx>;

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

The Accelerator Concept

Grid navigation

- The Accelerator provides the means to navigate the grid:

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

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

```
// get thread index on the block
auto blockThreadIdx = getIdx<Block, Threads>(acc);
```

```
// get number of blocks on the grid
auto gridBlockExtent = getWorkDiv<Grid, Blocks>(acc);
```

```
// get number of threads on the block
auto blockThreadExtent = getWorkDiv<Block, Threads>(acc);
```

The Accelerator Concept

Memory management and synchronization

- 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 = declareSharedVar<int, __COUNTER__>(acc);
```

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

- It also enables synchronization on the block level:

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

```
// synchronize some threads within the block and evaluate predicate
syncBlockThreadsPredicate(acc, predicate);
```

The Accelerator Concept

Device-side functions

- Internally, the accelerator maps all device-side functions to their native counterparts
- Device-side functions require the accelerator as first argument:
  - `sqrt(acc, /* … */);` (Math functions)
  - `atomicOp<AtomicOr>(acc, /* … */, hierarchy::Grids);` (Atomics)
  - `rand::distribution::createNormalReal<float>(acc);` (Random-number generation)
  - `clock(acc);` (Clock cycles)

The Device Concept

alpaka Devices

- alpaka Devices represent physical devices
- Determined by programmer’s Accelerator choice
- Easy management of physical devices

```
/* Chosen by programmer */
using Acc = AccGpuHipRt<Dim, Idx>;

/* Return number of HIP GPU devices */
auto const numDevs = getDevCount<Acc>();

/* Return the first entry from vector of HIP GPU devices */
auto myDev = getDevByIdx<Acc>(0u);

/* Return list of all HIP GPU devices */
auto devs = getDevs<Acc>();
```

The Device Concept

Devices and hardware

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

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

- Provides the means for device management:

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

- Encapsulates back-end device:

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

The Device Concept

alpaka Devices and the Accelerator concept

- Device and Accelerator are different concepts!
- An alpaka Accelerator is an abstract view of all physical devices (for the chosen back-end)
  - Kernel POV: thread state, device functions, memory management, synchronization
  - Host POV: meta-parameter for overall abstraction
- An alpaka Device is a representation of exactly one physical device
  - Device information
  - Device management

The Queue Concept

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 = property::NonBlocking;

// Define queue type
using MyQueue = Queue<Acc, QueueProperty>;

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

The Queue Concept

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Queue operations

- Queues execute Tasks (see next slide):

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

- Check for completion:

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

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

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

The Queue Concept

Tasks and Events

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

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

- Order of tasks in different queues is unspecified

```
enqueue(queueA, task1);
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);
enqueue(queueA, myEvent);
wait(queueB, myEvent); // queueB will only resume after queueA reached myEvent
```

The Queue Concept

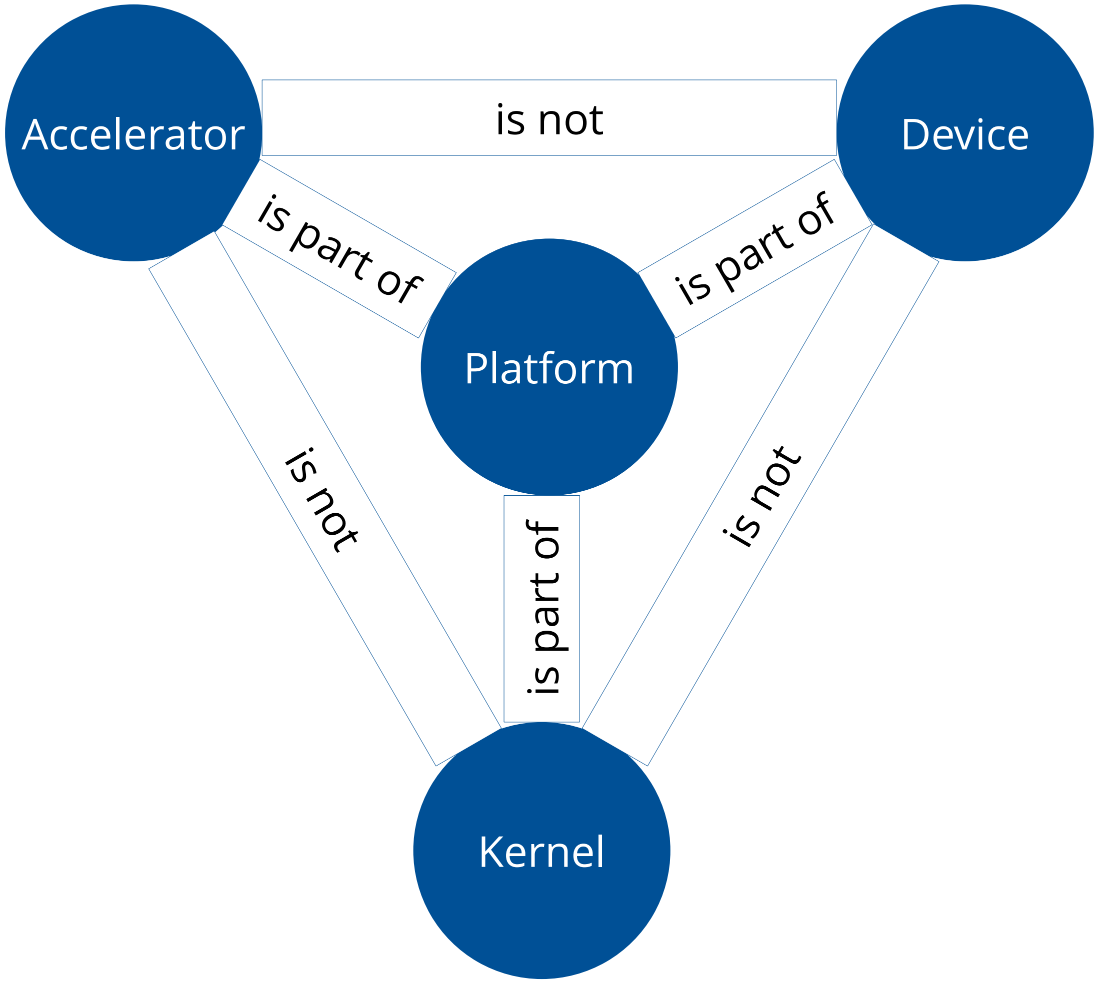
Setting up Accelerator, Device and Queue

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

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

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

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

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

The Platform Concept

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



The Platform Concept

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Changing the target platform

```
using namespace alpaka;

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

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

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

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

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

The Platform Concept

What alpaka does for you

- Configuration with standalone headers:
  - Enables chosen back-ends for your system
- After changing the Accelerator:
  - Back-end switched automatically
  - All Queue operations will be executed on associated devices

The Platform Concept

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

What you have to do for alpaka

- Standalone mode: Handle back-end dependencies and compiler flags
- Device side: Make no assumptions about your hardware!
  - Program your Kernels as abstract and portably as possible
  - Use the Accelerator for device-side operations
  - Kernels are instantiated for a specific platform at compile-time
  - This is what the Accelerator template parameter is for!

```
template <typename Acc>
ALPAKA_FN_ACC void operator()(Acc const & acc, /* … */) const;
```

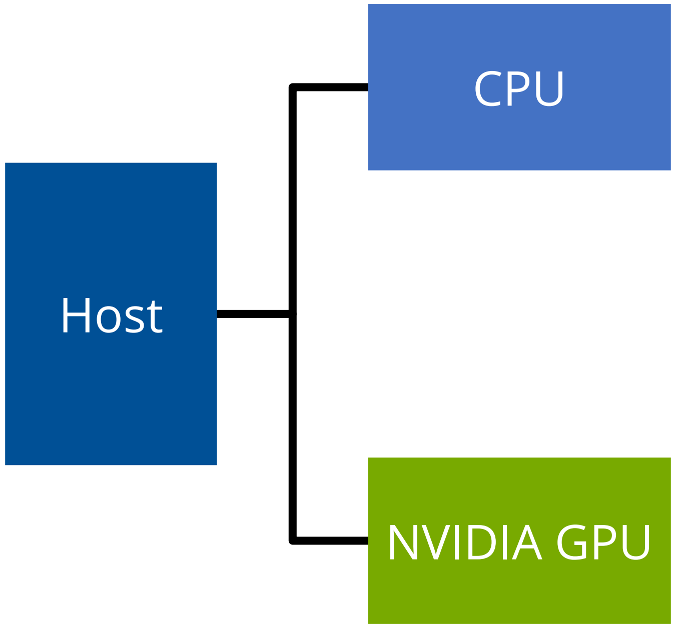
- Host side: Know your hardware!
  - Use Devices for management of physical devices
  - Adapt the work division (Blocks per Grid, Threads per Block, elements per Thread) to your hardware and problem size

Programming Heterogeneous Systems

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Using multiple Platforms

- 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 = AccCpuOmp2Blocks<Dim, Idx>;
using AccGpu = AccGpuCudaRt<Dim, Idx>;

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

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

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

Programming Heterogeneous Systems

Communication

- 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 = allocBuf<float, Idx>(devCpu, extent);
auto bufGpu = allocBuf<float, Idx>(devGpu, extent);

/* Initialization … */

// Copy buffer from CPU to GPU - destination comes first
memcpy(gpuQueue, bufGpu, bufCpu, extent);

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

// Copy results back to CPU and wait for completion
memcpy(gpuQueue, bufCpu, bufGpu, extent);

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

Programming Heterogeneous Systems

Heterogeneous programming with alpaka

- alpaka gives you access to all of your system's computation resources
- alpaka eases programming for different device types
- alpaka enables simple data transfers between different devices
- alpaka makes your code reusable
- alpaka makes your code portable

Write once, scale everywhere!

Heterogeneous Programming With the Caravan Ecosystem

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

I already have a CUDA program. Do I really need to port everything?

- No. Try our CUDA portability layer *cupla*.
- Kernels need to be ported to alpaka-style kernels
- `cudaApiCall()` becomes `cuplaApiCall()`
- **https://github.com/alpaka-group/cupla**

Heterogeneous Programming With the Caravan Ecosystem

How can I easily switch between different memory layouts?

- Example: From array-of-struct to struct-of-array and back
- Problem: Changing memory layout requires changing of algorithm
- Solution: LLAMA
- **https://github.com/alpaka-group/llama**

Heterogeneous Programming With the Caravan Ecosystem

But I just want to do transform & reduce!

- Solution: vikunja
- More standard algorithms planned soon
- **https://github.com/alpaka-group/vikunja**

CASUS

CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

TECHNISCHE UNIVERSITÄT DRESDEN

CBG Max Planck Institute of Molecular Cell Biology and Genetics

UFZ HELMHOLTZ Centre for Environmental Research

Uniwersytet Wrocławski

SPONSORED BY THE

Federal Ministry of Education and Research

STAATSMINISTERIUM FÜR WISSENSCHAFT KULTUR UND TOURISMUS

Freistaat SACHSEN