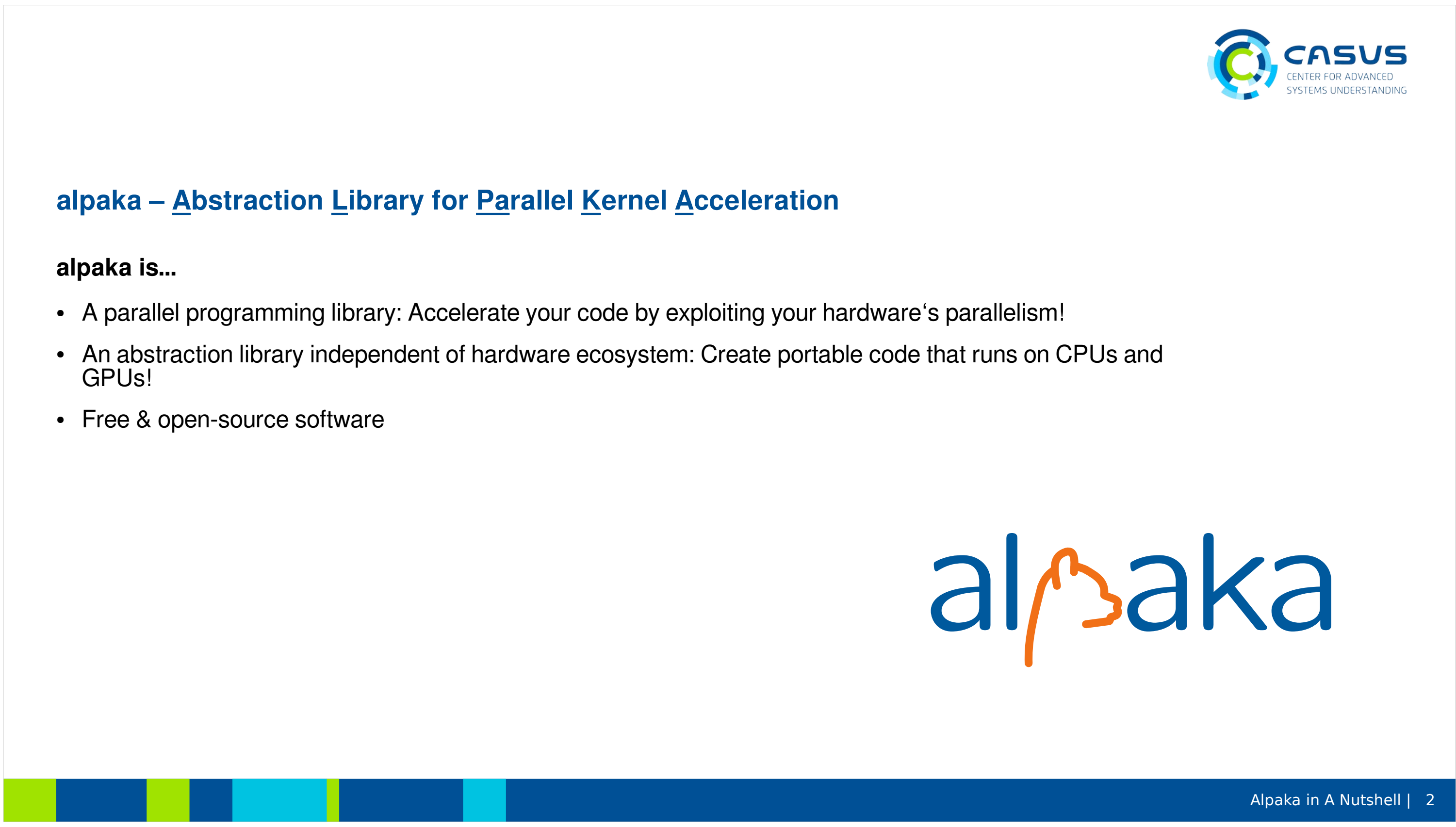
www.casus.scien
ce

Plasma-PEPSC Webinar
28 May 2024

alpaka Parallel Programming Library

Hello everyone. Thank you for participating. In this presentation I will initially introduce alpaka generally. Then I will go over an alpaka program and explain it step by step. Lastly I will talk about some performance tests and the contributors of alpaka.



alpaka – Abstraction Library for Parallel Kernel Acceleration

alpaka is…

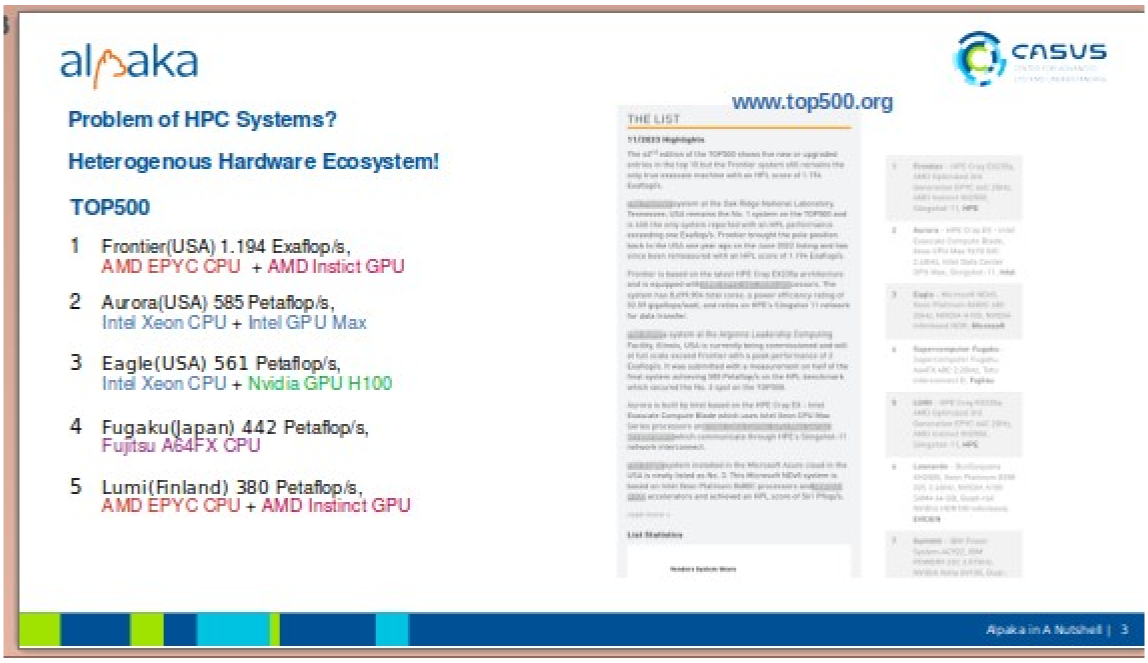
- A parallel programming library: Accelerate your code by exploiting your hardware‘s parallelism!
- An abstraction library independent of hardware ecosystem: Create portable code that runs on CPUs and GPUs!
- Free & open-source software

Alpaka is a parallel programming library. You can accelerate your code by exploiting your hardware’s paralelism. It is an abstraction library independent of hardware ecosystem.

With alpaka you can create create portable code which runs on different GPUs, and on CPUs.

Alpaka is free and open-source.

And lastly it is yet another library using an animal as logo.


alpaka

CASUS

Problem of HPC Systems?

Heterogenous Hardware Ecosystem!

TOP500

1. Frontier(USA) 1.194 Exaflop/s, AMD EPYC CPU + AMD Instict GPU
2. Aurora(USA) 585 Petaflop/s, Intel Xeon CPU + Intel GPU Max
3. Eagle(USA) 561 Petaflop/s, Intel Xeon CPU + Nvidia GPU H100
4. Fugaku(Japan) 442 Petaflop/s, Fujitsu A64FX CPU
5. Lumi(Finland) 380 Petaflop/s, AMD EPYC CPU + AMD Instinct GPU

www.top500.org

THE LIST

11/2023 Highlights

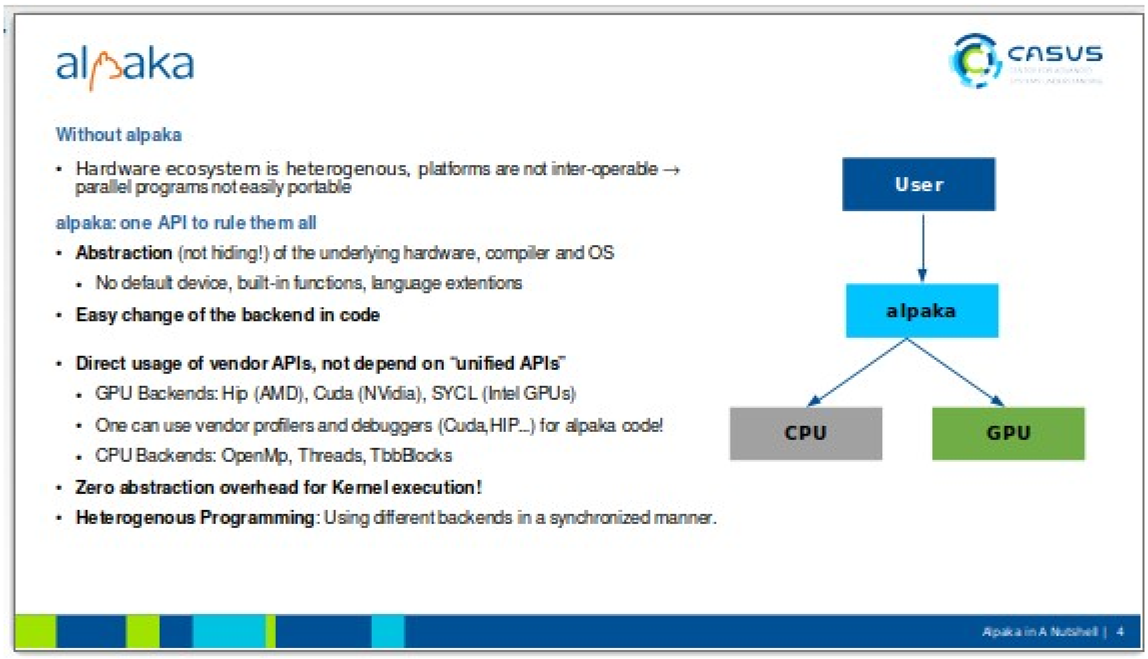
[illegible]

Alpaka in A Nutshell | 3

If we just go over the list of top HPC systems, we can see that hardware ecosystem is quite heterogenous.

The first one in the list, Frontier in USA is using AMD GPUs, Aurora is using Intel GPUs, the third one Eagle is using Nvidia GPUs, Fugaku in Japan is using CPUs which are actually CPUs with ARM architecture and Lumi in Finland is using AMD GPUs.

And these different vendors propose different programming APIs. Even they are using different colors in their logos and publications.



alpaka

Without alpaka

- Hardware ecosystem is heterogenous, platforms are not inter-operable → parallel programs not easily portable

alpaka: one API to rule them all

- **Abstraction** (not hiding!) of the underlying hardware, compiler and OS
  - No default device, built-in functions, language extentions
- **Easy change of the backend in code**
- **Direct usage of vendor APIs, not depend on "unified APIs"**
  - GPU Backends: Hip (AMD), Cuda (NVidia), SYCL (Intel GPUs)
  - One can use vendor profilers and debuggers (Cuda,HIP...) for alpaka code!
  - CPU Backends: OpenMp, Threads, TbbBlocks
- ***Zero abstraction overhead for Kernel execution!***
- **Heterogenous Programming**: Using different backends in a synchronized manner.

Hence currently HPC Platforms are not interoperable, or in other words programs are not portable.

Alpaka provides one API to support all different GPUs and CPU beckends.

Abstraction (but not hiding!) of the underlying hardware, compiler and OS is the main approach of Alpaka.

Alpaka does not have default device, built-in functions, language extentions, default stream like in cuda

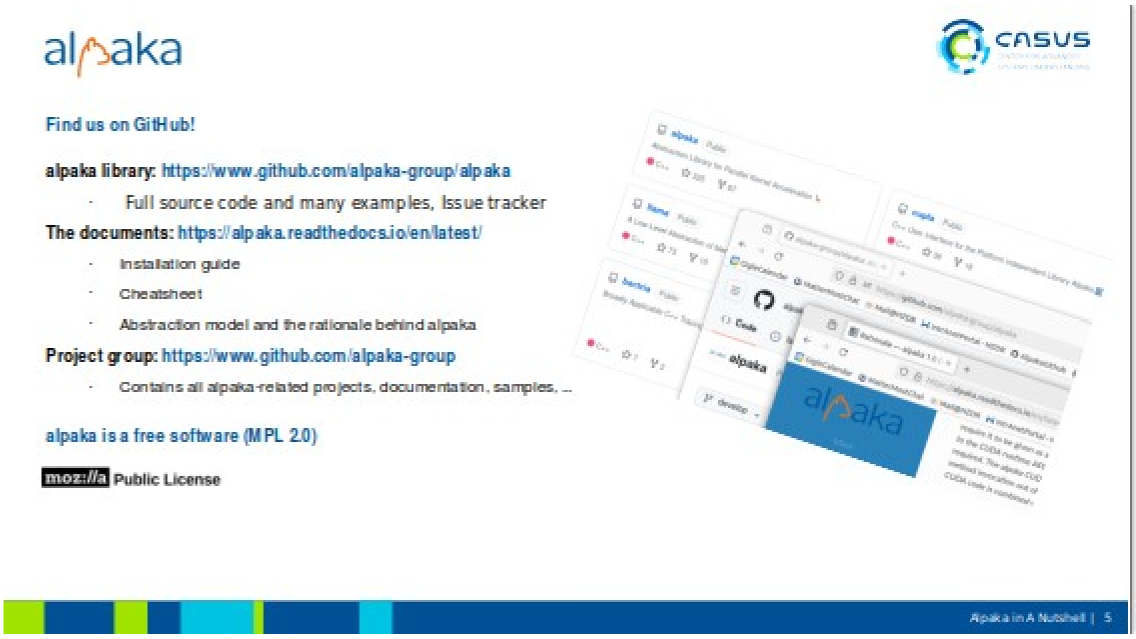
It is Easy to change the backend in code

Alpaka code directly uses vendor APIs, does not depend on "unified APIs". Produces the same code that a vendor API would generate. Hence alpaka has Zero abstraction overhead for Kernel execution!

Supported GPU Backends are Hip (AMD), Cuda (NVidia), SYCL (Intel GPUs)

alpaka user can use vendor profilers and debuggers (Cuda,HIP...) for his alpaka code!

Supported CPU Backends: OpenMp, Threads, TbbBlocks   **Heterogenous Programming**: Using different backends in a synchronized manner


alpaka

Find us on GitHub!

alpaka library: https://www.github.com/alpaka-group/alpaka
- Full source code and many examples, Issue tracker

The documents: https://alpaka.readthedocs.io/en/latest/
- Installation guide
- Cheatsheet
- Abstraction model and the rationale behind alpaka

Project group: https://www.github.com/alpaka-group
- Contains all alpaka-related projects, documentation, samples, ...

alpaka is a free software (MPL 2.0)

moz://a Public License

You can Find alpaka on GitHub!

The Github includes Full source code and many examples, and an Issue tracker

The documents pages at readthedocs includes:

-Installation guide

-Cheatsheet

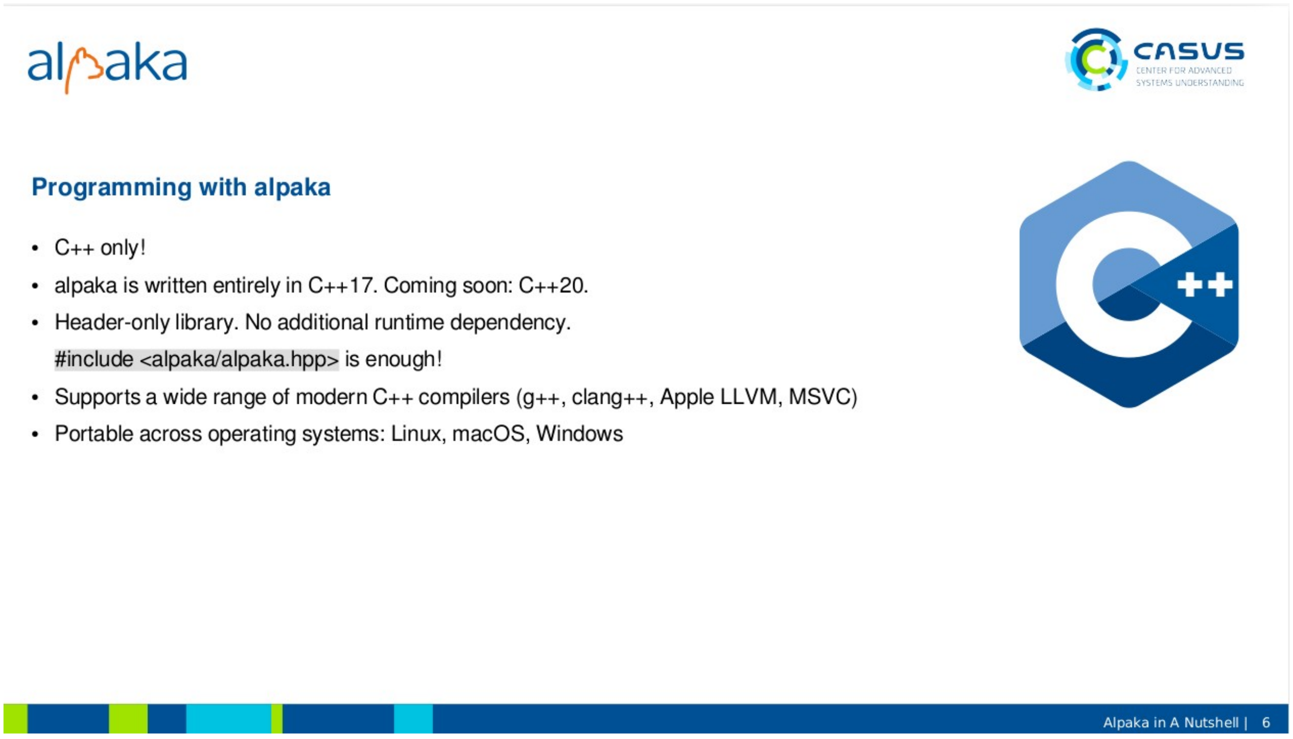
-Abstraction model and the rationale behind alpaka

Alpaka Project group: https://www.github.com/alpaka-group

Contains all alpaka-related projects, documentation, samples, …

Among those softwares:
- cupla easy porting from cuda,
- vikunja is for high level algorithms.

alpaka is a free software (MPL 2.0)



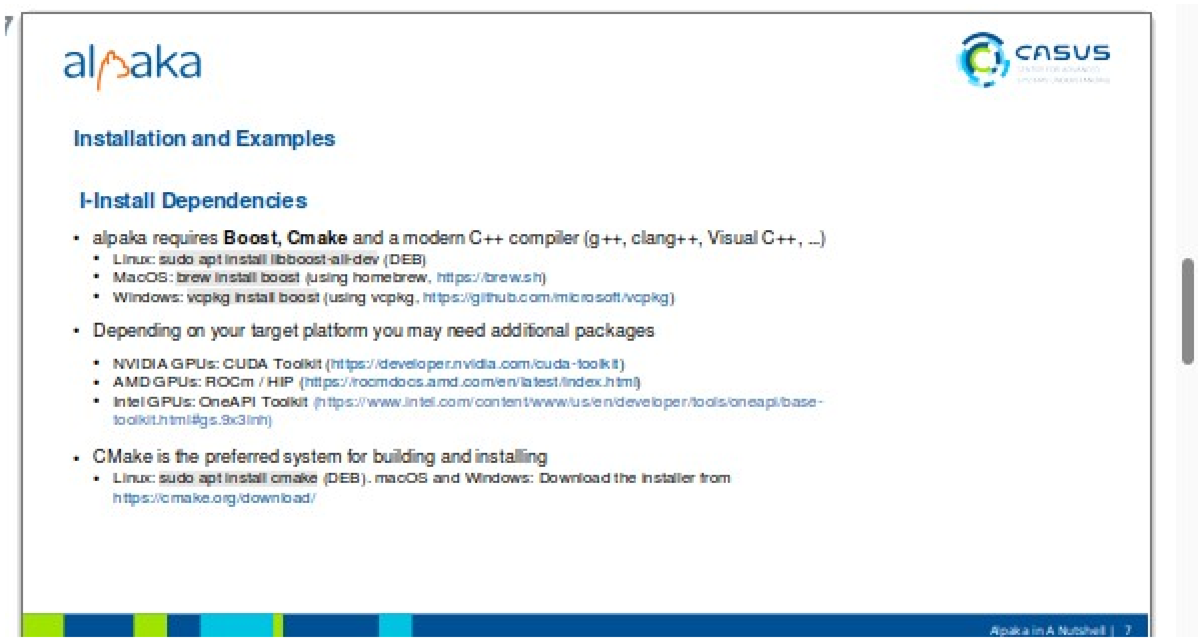
alpaka

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Programming with alpaka

- C++ only!
- alpaka is written entirely in C++17. Coming soon: C++20.
- Header-only library. No additional runtime dependency.
  `#include <alpaka/alpaka.hpp>` is enough!
- Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MSVC)
- Portable across operating systems: Linux, macOS, Windows

Alpaka is a library for C++. and it is written entirely in C++17. In a short time we will be using C++20 features and compiling on : C++20.
It is a Header-only library. No additional runtime dependency is used. Only Compile time dependency is Boost.
Including the header file alpaka/alpaka.hpp to the cpp code would be enough!
Alpaka Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MSVC)
and it is Portable across operating systems: Linux, macOS, Windows


alpaka

Installation and Examples

I-Install Dependencies

- alpaka requires **Boost, Cmake** and a modern C++ compiler (g++, clang++, Visual C++, ...)
  - Linux: `sudo apt install libboost-all-dev` (DEB)
  - MacOS: `brew install boost` (using homebrew, https://brew.sh)
  - 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)
  - Intel GPUs: OneAPI Toolkit (https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html#gs.9x3inh)
- CMake is the preferred system for building and installing
  - Linux: `sudo apt install cmake` (DEB). macOS and Windows: Download the installer from https://cmake.org/download/

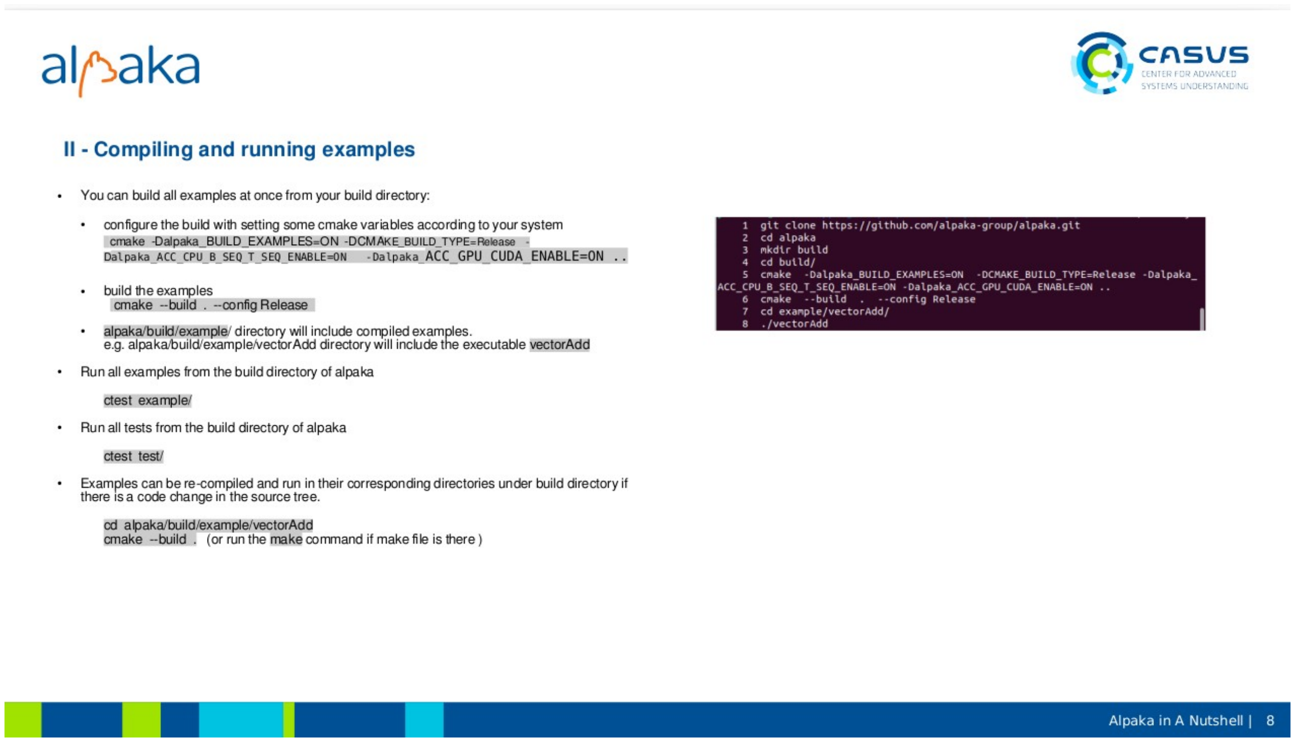
Alpaka in A Nutshell | 7

Installation of Alpaka and Building Examples
Initially we need to install dependencies

alpaka requires Boost as a compile time dependency for compilation. For the configuration of the build system Cmake and for the compilation a C++ compiler (g++, clang++, Visual C++, …) is needed.

Depending on your target platform you may need additional packages of Cuda, Rocm or Intel OneAPI toolkits.

CMake is the preferred system for building and installing



II - Compiling and running examples

- You can build all examples at once from your build directory:
  - configure the build with setting some cmake variables according to your system
    `cmake -Dalpaka_BUILD_EXAMPLES=ON -DCMAKE_BUILD_TYPE=Release -Dalpaka_ACC_CPU_B_SEQ_T_SEQ_ENABLE=ON -Dalpaka_ACC_GPU_CUDA_ENABLE=ON ..`
  - build the examples
    `cmake --build . --config Release`
  - alpaka/build/example/ directory will include compiled examples.
    e.g. alpaka/build/example/vectorAdd directory will include the executable vectorAdd
- Run all examples from the build directory of alpaka

  `ctest example/`
- Run all tests from the build directory of alpaka

  `ctest test/`
- Examples can be re-compiled and run in their corresponding directories under build directory if there is a code change in the source tree.

  `cd alpaka/build/example/vectorAdd`
  `cmake --build .` (or run the make command if make file is there)

```
git clone https://github.com/alpaka-group/alpaka.git
cd alpaka
mkdir build
cd build/
cmake  -Dalpaka_BUILD_EXAMPLES=ON  -DCMAKE_BUILD_TYPE=Release -Dalpaka_
ACC_CPU_B_SEQ_T_SEQ_ENABLE=ON -Dalpaka_ACC_GPU_CUDA_ENABLE=ON ..
cmake  --build  .  --config Release
cd example/vectorAdd/
./vectorAdd
```

Alpaka in A Nutshell | 8

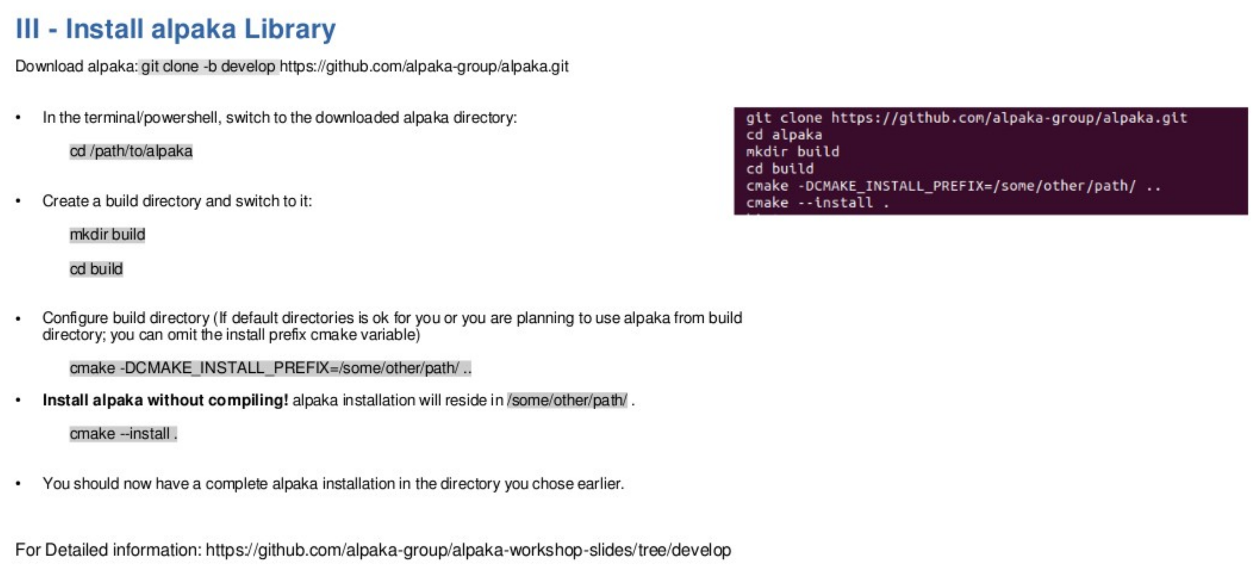
After installing boost and cmake and compilation tools We don't need to install alpaka files to compile examples. We can just directly compile and run.

In compiling examples or any program using alpaka; Setting cmake variables are important. The user needs to configure the build with setting the cmake variables according to her/his system:

cmake -Dalpaka_BUILD_EXAMPLES=ON -DCMAKE_BUILD_TYPE=Release -Dalpaka_ACC_CPU_B_SEQ_T_SEQ_ENABLE=ON
-Dalpaka_ ACC_GPU_CUDA_ENABLE=ON

These backend settings doesn't mean the user has to use these backends in the code but means they are available and can be used by the user. (// Example selects which one?)

After building, Runnin all examples by **ctest example** command or all tests by **ctest test** command is possible.



III - Install alpaka Library

Download alpaka: git clone -b develop https://github.com/alpaka-group/alpaka.git

- In the terminal/powershell, switch to the downloaded alpaka directory:

  cd /path/to/alpaka

- Create a build directory and switch to it:

  mkdir build

  cd build

- Configure build directory (If default directories is ok for you or you are planning to use alpaka from build directory; you can omit the install prefix cmake variable)

  cmake -DCMAKE_INSTALL_PREFIX=/some/other/path/ ..

- **Install alpaka without compiling!** alpaka installation will reside in /some/other/path/ .

  cmake --install .

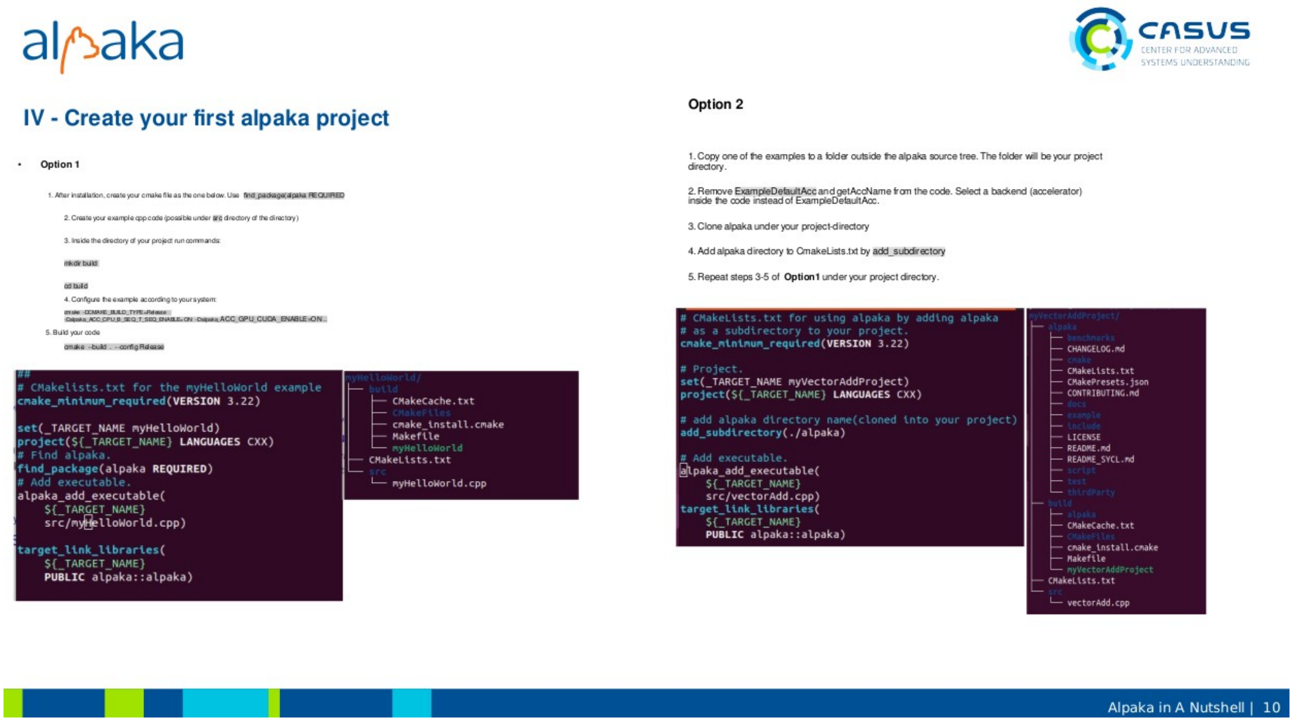
- You should now have a complete alpaka installation in the directory you chose earlier.

```
git clone https://github.com/alpaka-group/alpaka.git
cd alpaka
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/some/other/path/ ..
cmake --install .
```

For Detailed information: https://github.com/alpaka-group/alpaka-workshop-slides/tree/develop

Installation of alpaka is quite strait-forward. Compilation is not needed, just 2 steps are needed: configuring and installing.

Notice that you don’t need to set cmake variables depending on our system for installation; because those variables will be just before the compilation of your project. Namely configuring your project.



IV - Create your first alpaka project

- **Option 1**

1. After installation, create your cmake file as the one below. Use find_package(alpaka REQUIRED

2. Create your example cpp code (possible under src directory of the directory)

3. Inside the directory of your project run commands:

mkdir build

cd build

4. Configure the example according to your system:

cmake -DCMAKE_BUILD_TYPE=Release -Dalpaka_ACC_CPU_B_SEQ_T_SEQ_ENABLE=ON -Dalpaka_ACC_GPU_CUDA_ENABLE=ON..

5. Build your code

cmake --build . --config Release

```
##
# CMakelists.txt for the myHelloWorld example
cmake_minimum_required(VERSION 3.22)

set(_TARGET_NAME myHelloWorld)
project(${_TARGET_NAME} LANGUAGES CXX)
# Find alpaka.
find_package(alpaka REQUIRED)
# Add executable.
alpaka_add_executable(
    ${_TARGET_NAME}
    src/myHelloWorld.cpp)

target_link_libraries(
    ${_TARGET_NAME}
    PUBLIC alpaka::alpaka)
```

```
myHelloWorld/
├── build
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   ├── Makefile
│   └── myHelloWorld
├── CMakeLists.txt
└── src
    └── myHelloWorld.cpp
```

Option 2

1. Copy one of the examples to a folder outside the alpaka source tree. The folder will be your project directory.

2. Remove ExampleDefaultAcc and getAccName from the code. Select a backend (accelerator) inside the code instead of ExampleDefaultAcc.

3. Clone alpaka under your project-directory

4. Add alpaka directory to CmakeLists.txt by add_subdirectory

5. Repeat steps 3-5 of **Option1** under your project directory.

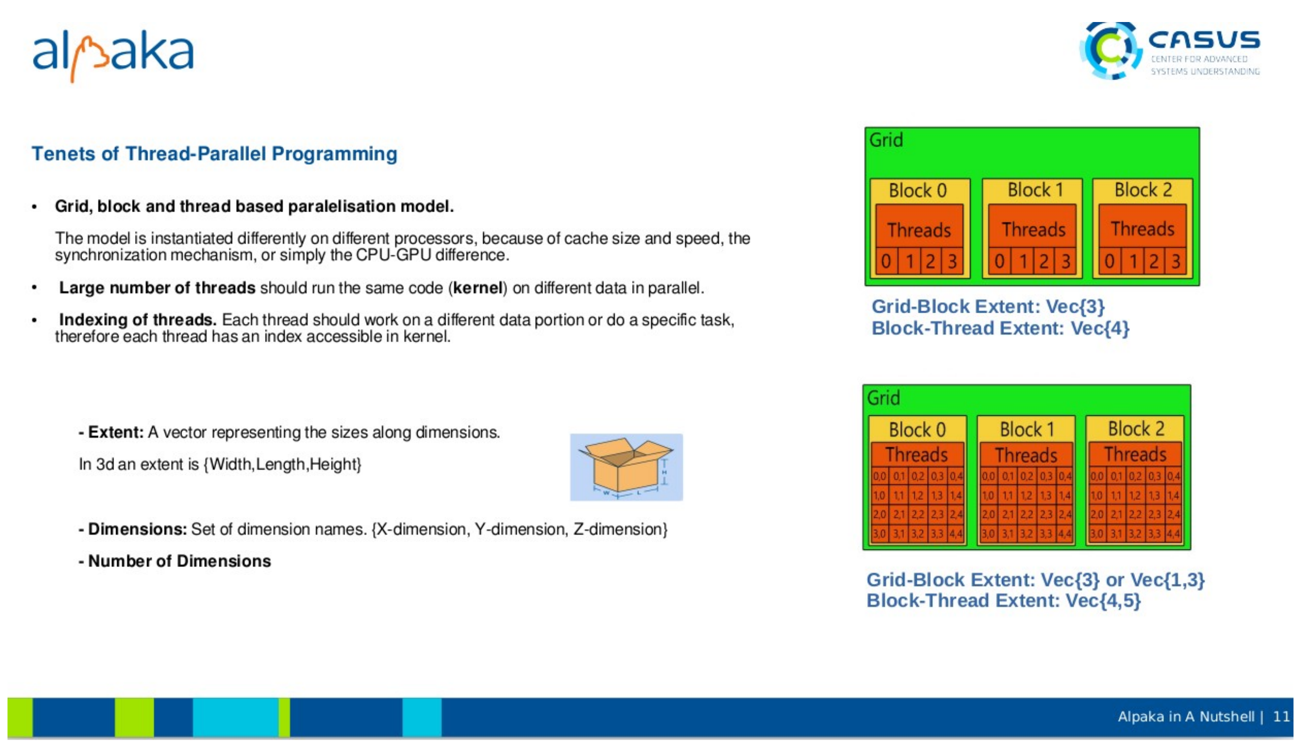
```
# CMakeLists.txt for using alpaka by adding alpaka
# as a subdirectory to your project.
cmake_minimum_required(VERSION 3.22)

# Project.
set(_TARGET_NAME myVectorAddProject)
project(${_TARGET_NAME} LANGUAGES CXX)

# add alpaka directory name(cloned into your project)
add_subdirectory(./alpaka)

# Add executable.
alpaka_add_executable(
    ${_TARGET_NAME}
    src/vectorAdd.cpp)
target_link_libraries(
    ${_TARGET_NAME}
    PUBLIC alpaka::alpaka)
```

```
myVectorAddProject/
├── alpaka
│   ├── benchmarks
│   ├── CHANGELOG.md
│   ├── cmake
│   ├── CMakeLists.txt
│   ├── CMakePresets.json
│   ├── CONTRIBUTING.md
│   ├── docs
│   ├── example
│   ├── include
│   ├── LICENSE
│   ├── README.md
│   ├── README_SYCL.md
│   ├── script
│   ├── test
│   └── thirdParty
├── build
│   ├── alpaka
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   ├── Makefile
│   └── myVectorAddProject
├── CMakeLists.txt
└── src
    └── vectorAdd.cpp
```

Creating your alpaka project can be achieved in 2 ways. First way is, creating your **cmakelists** file and your code file in a directory you chose.

Secondly you can just copy one of the examples as a directory and change the code a little bit.

The important point for both cases is that. Select all possible accelerators while configuring the build tree by cmake, that means, user sets the corresponding cmake variables to activate accelerators during the configuration by cmake.



Tenets of Thread-Parallel Programming

- **Grid, block and thread based paralelisation model.**

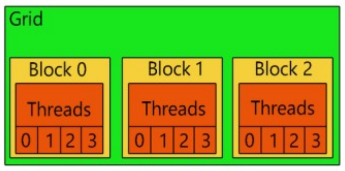
  The model is instantiated differently on different processors, because of cache size and speed, the synchronization mechanism, or simply the CPU-GPU difference.
- **Large number of threads** should run the same code (**kernel**) on different data in parallel.
- **Indexing of threads.** Each thread should work on a different data portion or do a specific task, therefore each thread has an index accessible in kernel.

- Extent: A vector representing the sizes along dimensions.

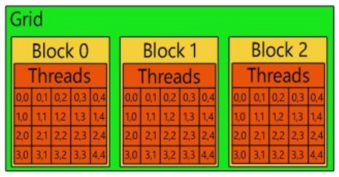
In 3d an extent is {Width,Length,Height}

- Dimensions: Set of dimension names. {X-dimension, Y-dimension, Z-dimension}

- Number of Dimensions



Grid-Block Extent: Vec{3}
Block-Thread Extent: Vec{4}



Grid-Block Extent: Vec{3} or Vec{1,3}
Block-Thread Extent: Vec{4,5}

Before moving on a small alpaka code I would like to talk about basic concepts of thread-parallel programming.

Alpaka uses Grid-block-thread based paralelisation model. The model is instantiated differently on different processors, because of cache size and the cache speed, the synchronization mechanism, or simply the difference between CPU-GPUs. By using grid-block-thread abstraction the execution can be optimally adapted to the available hardware.

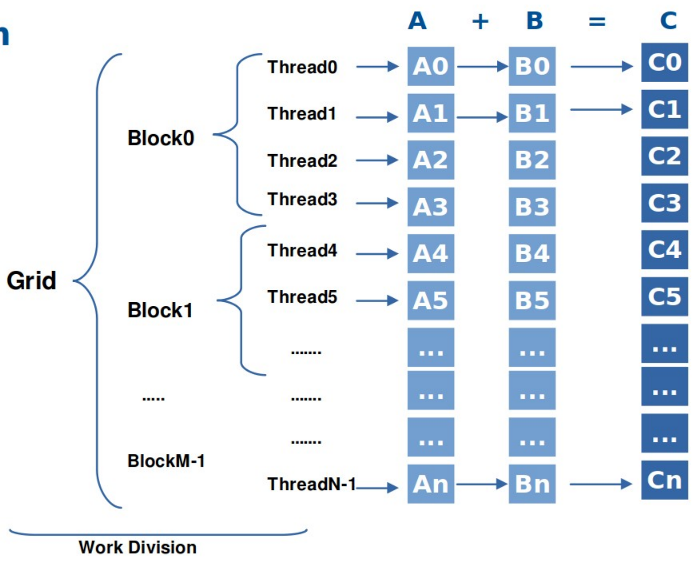
Secondly the assumption is that Large number of threads should run the same code (kernel) on different data in a parallel manner.

Lastly; Indexing of threads is needed. Each thread should work on a different data portion or do a specific task,
therefore each thread has an index accessible in kernel.

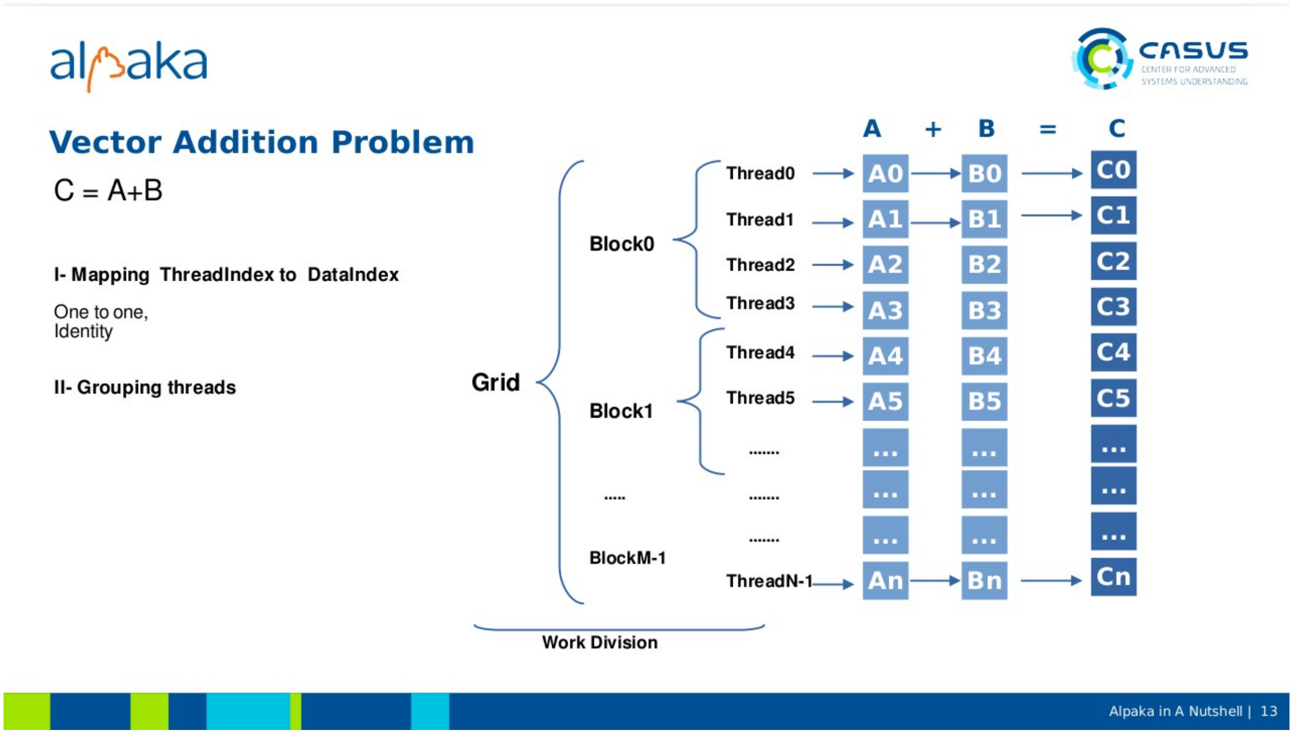
There are 3 terms I would like to describe. The extent means the sizes along each dimensions. In 3d for example extent is a 3 item vector of {width,length and height}. The term Dimensions means “set of dimension names” although in daily english we use dimensions as the extent.
Lastly number of dimensions is the size of the extent vector.

Vector Addition Problem

C = A+B



Lets assume that we want to sum to vectors by utilising paralism of the hardware. We have 2 one dimensional vectors, vector A and vector B and we are going to calculate their sum C.



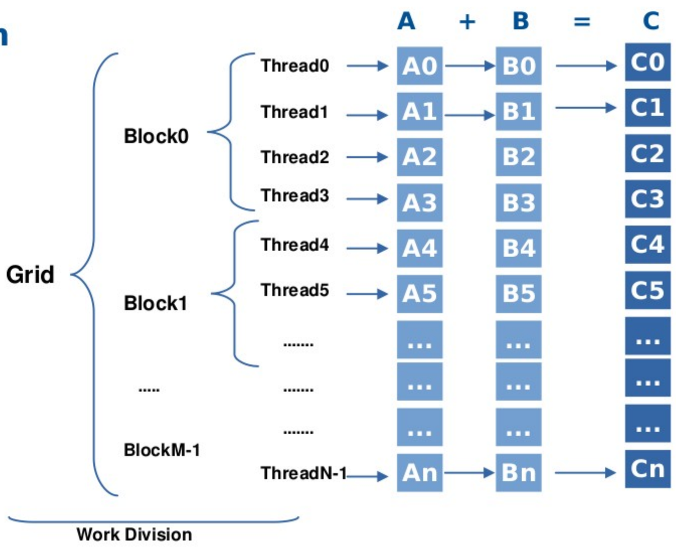
Vector Addition Problem

C = A+B

I- Mapping ThreadIndex to DataIndex

One to one,
Identity

II- Grouping threads

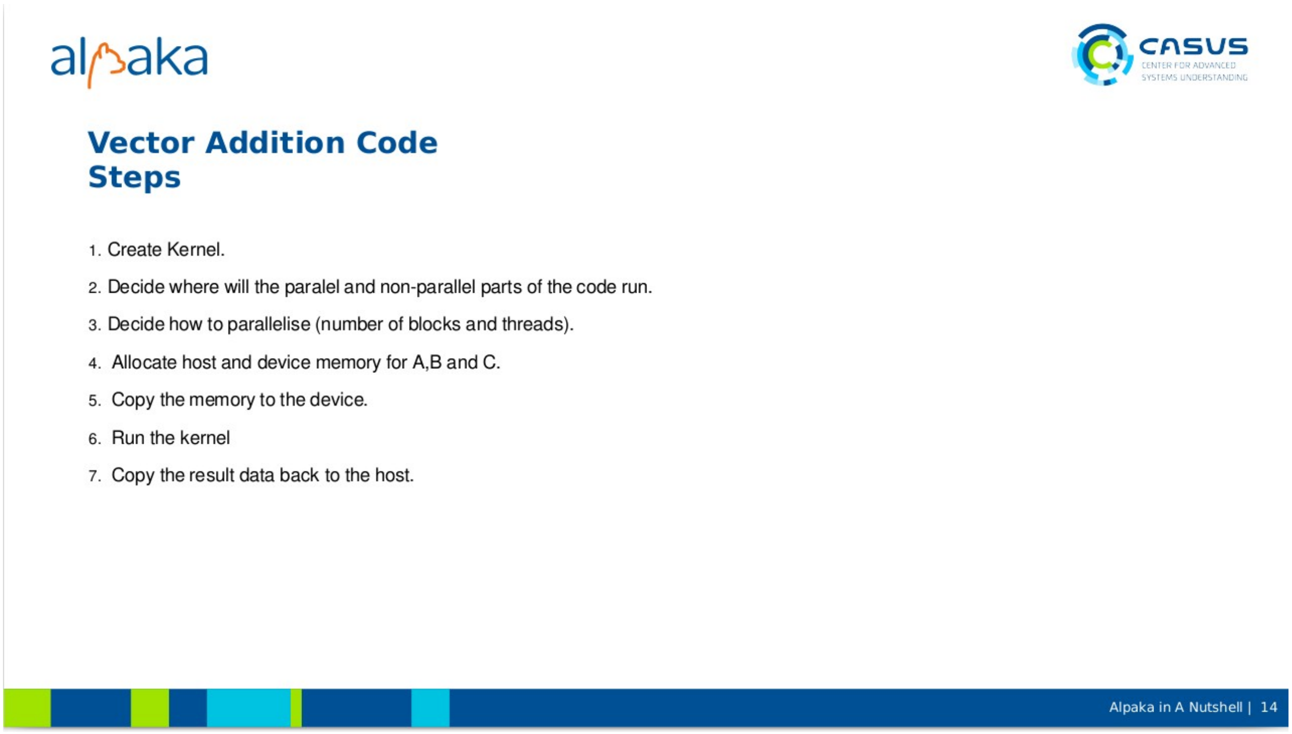


Typically for paralellisation we need to divide the calculation of vector C, into parallel summations.

That means we need to map threads to the data. In our solution as you can see on the graph a one-to-one mapping, actually an identity function, between thread indices and data indices is used.

Thread0 will sum A[0] and B[0] to find C[0], thread1 will sum A[1] and b[1] to find C[1] and so on.

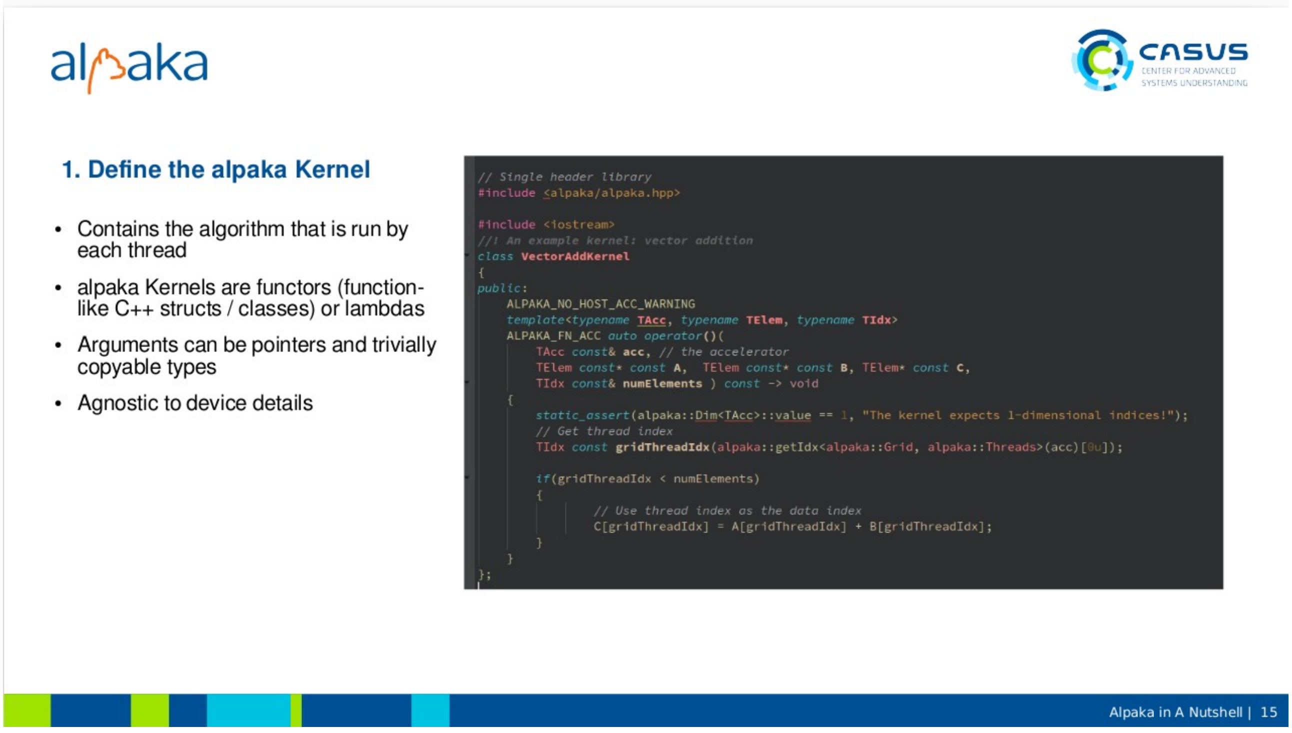
The Second issue is how to select or define the grid-block-thread paralelism or in other words determining the alpaka work division. In this representation we have M blocks in grid and each block has 4 threads.



Vector Addition Code Steps

1. Create Kernel.
2. Decide where will the paralel and non-parallel parts of the code run.
3. Decide how to parallelise (number of blocks and threads).
4. Allocate host and device memory for A,B and C.
5. Copy the memory to the device.
6. Run the kernel
7. Copy the result data back to the host.

From the coding perspective; the vector addition code should have these steps:
1. Create Kernel.
2. Decide where will the paralel and non-parallel parts of the code run.
3. Decide how to parallelise (number of blocks and threads).
4. Allocate host and device memory for A,B and C.
5. Copy the memory to the device.
6. Run the kernel
7. Copy the result data back to the host.


alpaka

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

1. Define the alpaka Kernel

- Contains the algorithm that is run by each thread
- alpaka Kernels are functors (function-like C++ structs / classes) or lambdas
- Arguments can be pointers and trivially copyable types
- Agnostic to device details

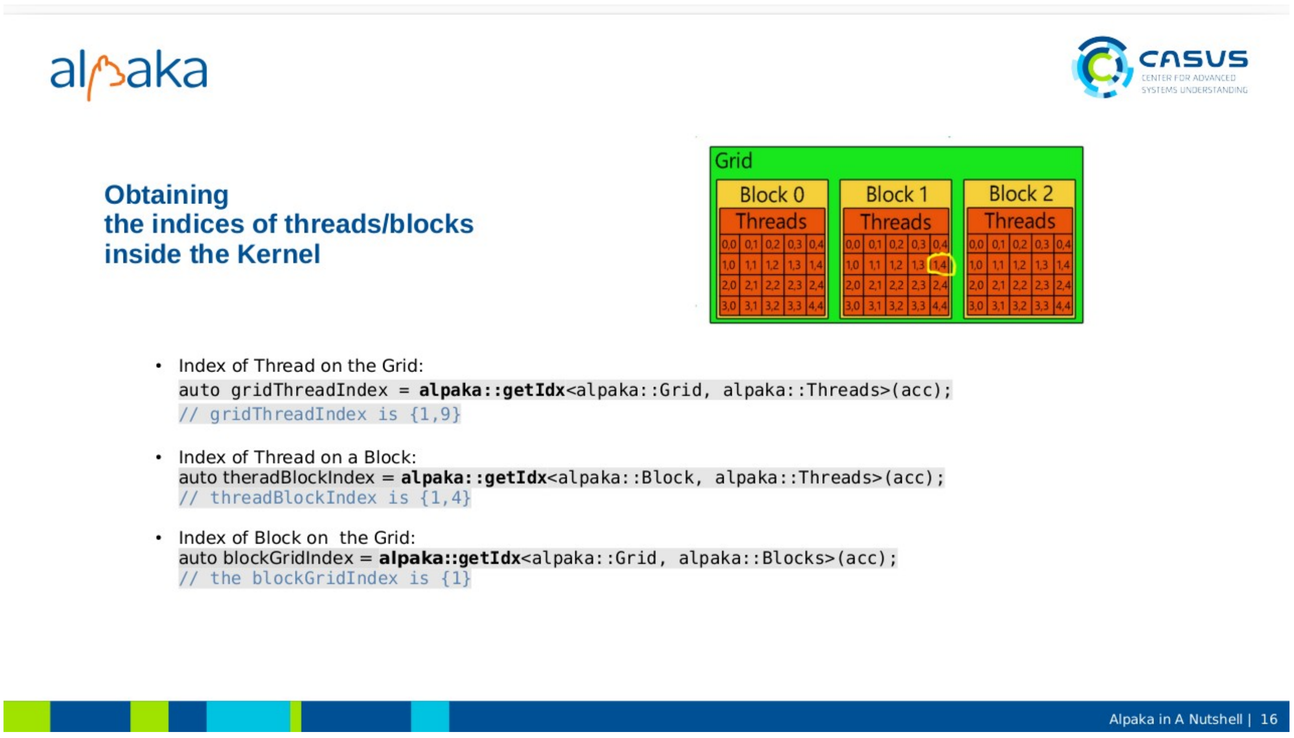
```cpp
// Single header library
#include <alpaka/alpaka.hpp>

#include <iostream>
//! An example kernel: vector addition
class VectorAddKernel
{
public:
    ALPAKA_NO_HOST_ACC_WARNING
    template<typename TAcc, typename TElem, typename TIdx>
    ALPAKA_FN_ACC auto operator()(
        TAcc const& acc, // the accelerator
        TElem const* const A,  TElem const* const B, TElem* const C,
        TIdx const& numElements ) const -> void
    {
        static_assert(alpaka::Dim<TAcc>::value == 1, "The kernel expects 1-dimensional indices!");
        // Get thread index
        TIdx const gridThreadIdx(alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u]);

        if(gridThreadIdx < numElements)
        {
                // Use thread index as the data index
                C[gridThreadIdx] = A[gridThreadIdx] + B[gridThreadIdx];
        }
    }
};
```

At the first step kernel is defined. Kernel contains code that is run by each thread. Alpaka kernels are functors, namely structs with specificly implemented function operators or lambdas.

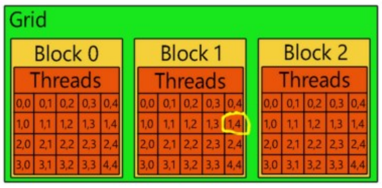
Arguments of the function operator can be pointers or trivially copyable types. You can put many pointers and built-in types in a struct and just pass the struct as a value for example.

Alpaka kernel is agnostic to device details.

As you see, Alpaka is low level and transparent. Abstraction usually associated with being high level but alpaka is low level in that sense. One can access to the thread index directly as in Hip or Cuda.



Obtaining the indices of threads/blocks inside the Kernel



- Index of Thread on the Grid:

```
auto gridThreadIndex = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc);
// gridThreadIndex is {1,9}
```

- Index of Thread on a Block:

```
auto theradBlockIndex = alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc);
// threadBlockIndex is {1,4}
```

- Index of Block on the Grid:

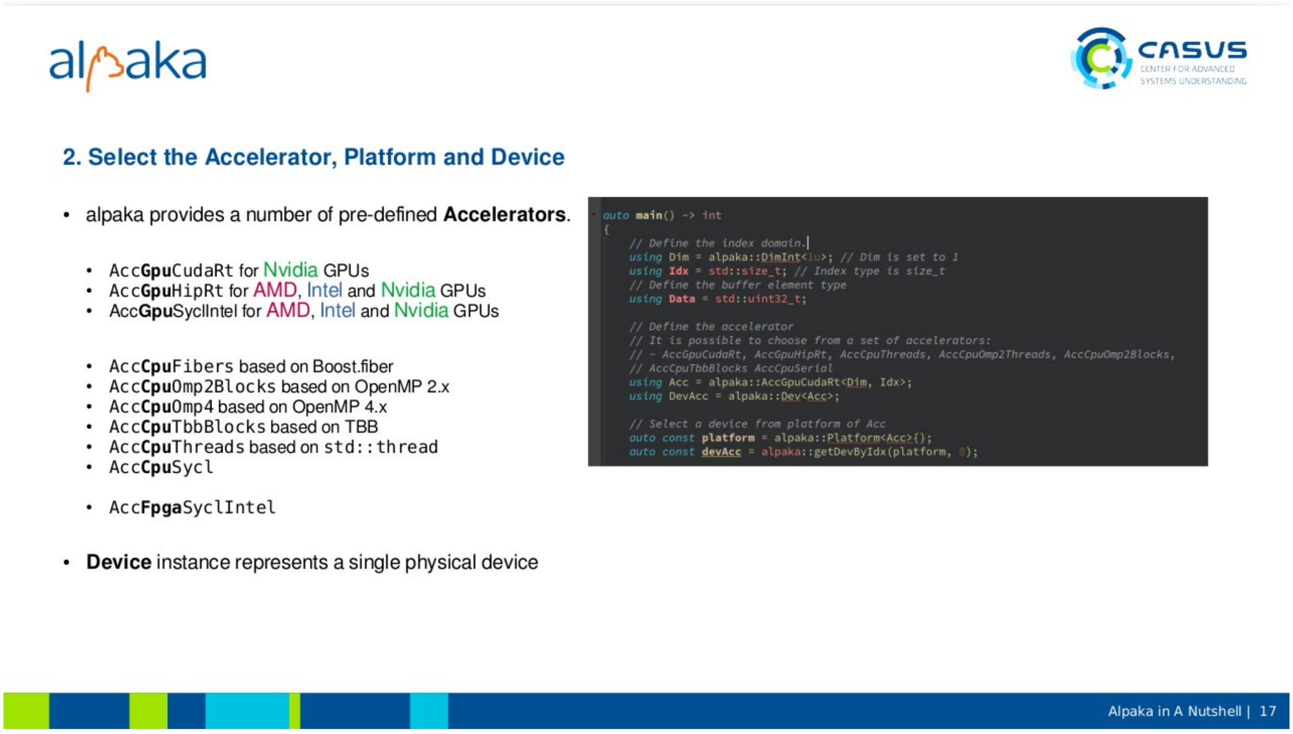
```
auto blockGridIndex = alpaka::getIdx<alpaka::Grid, alpaka::Blocks>(acc);
// the blockGridIndex is {1}
```

Obtaining the indices of threads or blocks is easy in alpaka kernel, thanks to usage of function templates.

Using getIdx function with different predetermined template arguments would be enough get the thread index in grid or in block. Or the index of block in the grid.

For eample the thread [1,4] of the block1 would have a grid index [1,9] because it is on the second row and the 10th column in the grid. Note that alpaka::Grid and alpaka::Threads types are used as template arguments.

Same thread will have a threadBlock index [1,4] which shows its coordinates wrt the block.



2. Select the Accelerator, Platform and Device

- alpaka provides a number of pre-defined **Accelerators**.
  - Acc**Gpu**CudaRt for Nvidia GPUs
  - Acc**Gpu**HipRt for AMD, Intel and Nvidia GPUs
  - Acc**Gpu**SyclIntel for AMD, Intel and Nvidia GPUs
  - Acc**Cpu**Fibers based on Boost.fiber
  - Acc**Cpu**Omp2Blocks based on OpenMP 2.x
  - Acc**Cpu**Omp4 based on OpenMP 4.x
  - Acc**Cpu**TbbBlocks based on TBB
  - Acc**Cpu**Threads based on std::thread
  - Acc**Cpu**Sycl
  - Acc**Fpga**SyclIntel
- **Device** instance represents a single physical device

```
auto main() -> int
{
    // Define the index domain.
    using Dim = alpaka::DimInt<1u>; // Dim is set to 1
    using Idx = std::size_t; // Index type is size_t
    // Define the buffer element type
    using Data = std::uint32_t;

    // Define the accelerator
    // It is possible to choose from a set of accelerators:
    // - AccGpuCudaRt, AccGpuHipRt, AccCpuThreads, AccCpuOmp2Threads, AccCpuOmp2Blocks,
    // AccCpuTbbBlocks AccCpuSerial
    using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
    using DevAcc = alpaka::Dev<Acc>;

    // Select a device from platform of Acc
    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);
```

Now we can design the code part that will NOT run in parallel, namely the code that will run on the host device which will just trigger the parallel part.

Initialy selecting the one of the predefined accelerator types according to your system is needed. There are 3 accelerator types for as the GPU backend, 6 types for the CPU backend and 1 type for the Fpga backend. The accelerator name actualy has 2 important parts first one is the device type (CPU or GPU) and the second part the specific API.

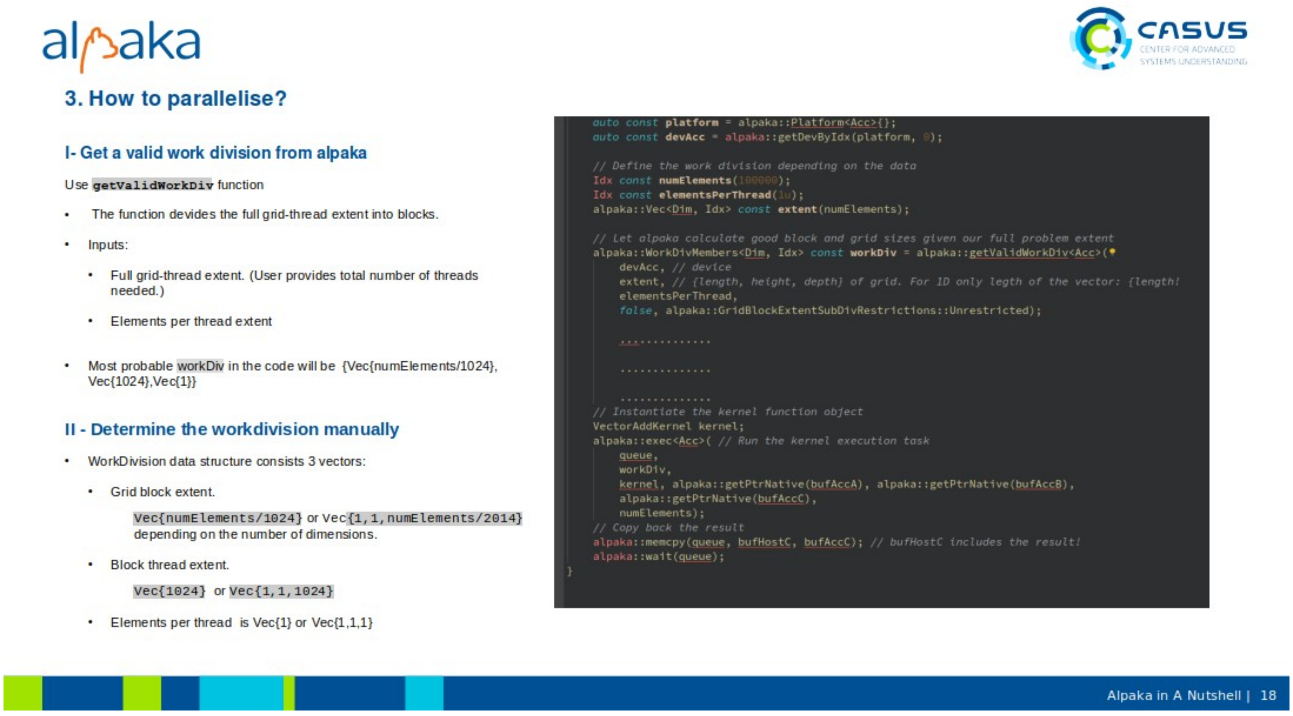
// text Accelerator is an abstraction that makes the kernel ‘device agnostic’.

// easily change the accelerator

Accelerator is a type that is only instantiated on Device not on the host.

Easy management of physical devices

Contains device information



3. How to parallelise?

I- Get a valid work division from alpaka

Use `getValidWorkDiv` function

- The function devides the full grid-thread extent into blocks.
- Inputs:
  - Full grid-thread extent. (User provides total number of threads needed.)
  - Elements per thread extent
- Most probable workDiv in the code will be {Vec{numElements/1024}, Vec{1024},Vec{1}}

II - Determine the workdivision manually

- WorkDivision data structure consists 3 vectors:
  - Grid block extent.

    `Vec{numElements/1024}` or Vec`{1,1,numElements/2014}` depending on the number of dimensions.
  - Block thread extent.

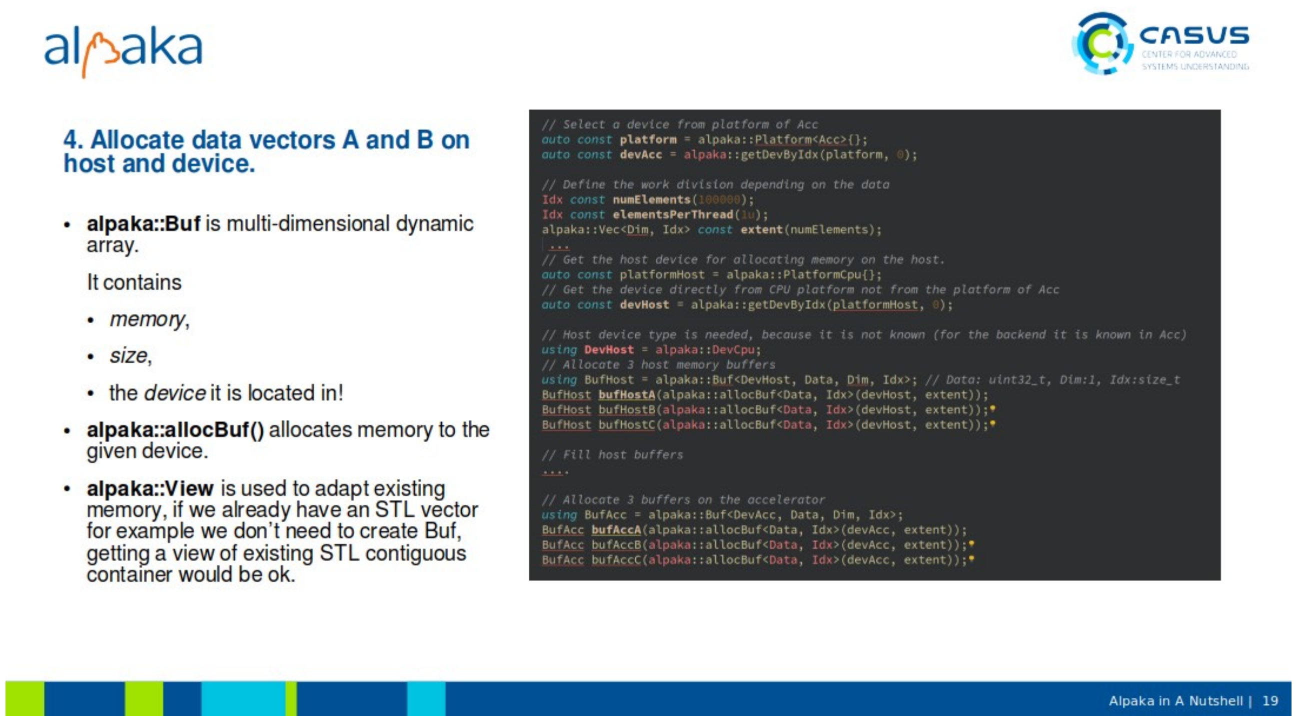
    `Vec{1024}` or `Vec{1,1,1024}`
  - Elements per thread is Vec{1} or Vec{1,1,1}

```cpp
    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);

    // Define the work division depending on the data
    Idx const numElements(100000);
    Idx const elementsPerThread(1u);
    alpaka::Vec<Dim, Idx> const extent(numElements);

    // Let alpaka calculate good block and grid sizes given our full problem extent
    alpaka::WorkDivMembers<Dim, Idx> const workDiv = alpaka::getValidWorkDiv<Acc>(
        devAcc, // device
        extent, // {length, height, depth} of grid. For 1D only legth of the vector: {length!
        elementsPerThread,
        false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

        ..............

        ..............

        ..............
    // Instantiate the kernel function object
    VectorAddKernel kernel;
    alpaka::exec<Acc>( // Run the kernel execution task
        queue,
        workDiv,
        kernel, alpaka::getPtrNative(bufAccA), alpaka::getPtrNative(bufAccB),
        alpaka::getPtrNative(bufAccC),
        numElements);
    // Copy back the result
    alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
    alpaka::wait(queue);
}
```

Alpaka in A Nutshell | 18

Paralelisation model or work division can be selected using an alpaka function GetValidWorkDiv, which takes the full grid-thread extent as argument and devides the given extent into blocks. It takes namely a massive box of threads without any subdivisions and devides thisbox of threads into blocks. The generated block size is inside the allowed limits which is usually 1024 threads per block for GPUs. Elements per thread extent is a sub index which could be assigned to each thread; if each thread needs a number of additional indices.

Secondly work division can be set by the user. Grid-block, block thread extent and thread-elem extent should be determined by the user.



4. Allocate data vectors A and B on host and device.

- **alpaka::Buf** is multi-dimensional dynamic array.

  It contains
  - *memory*,
  - *size*,
  - the *device* it is located in!
- **alpaka::allocBuf()** allocates memory to the given device.
- **alpaka::View** is used to adapt existing memory, if we already have an STL vector for example we don't need to create Buf, getting a view of existing STL contiguous container would be ok.

```
// Select a device from platform of Acc
auto const platform = alpaka::Platform<Acc>{};
auto const devAcc = alpaka::getDevByIdx(platform, 0);

// Define the work division depending on the data
Idx const numElements(100000);
Idx const elementsPerThread(1u);
alpaka::Vec<Dim, Idx> const extent(numElements);
 ...
// Get the host device for allocating memory on the host.
auto const platformHost = alpaka::PlatformCpu{};
// Get the device directly from CPU platform not from the platform of Acc
auto const devHost = alpaka::getDevByIdx(platformHost, 0);

// Host device type is needed, because it is not known (for the backend it is known in Acc)
using DevHost = alpaka::DevCpu;
// Allocate 3 host memory buffers
using BufHost = alpaka::Buf<DevHost, Data, Dim, Idx>; // Data: uint32_t, Dim:1, Idx:size_t
BufHost bufHostA(alpaka::allocBuf<Data, Idx>(devHost, extent));
BufHost bufHostB(alpaka::allocBuf<Data, Idx>(devHost, extent));
BufHost bufHostC(alpaka::allocBuf<Data, Idx>(devHost, extent));

// Fill host buffers
....

// Allocate 3 buffers on the accelerator
using BufAcc = alpaka::Buf<DevAcc, Data, Dim, Idx>;
BufAcc bufAccA(alpaka::allocBuf<Data, Idx>(devAcc, extent));
BufAcc bufAccB(alpaka::allocBuf<Data, Idx>(devAcc, extent));
BufAcc bufAccC(alpaka::allocBuf<Data, Idx>(devAcc, extent));
```

Alpaka in A Nutshell | 19

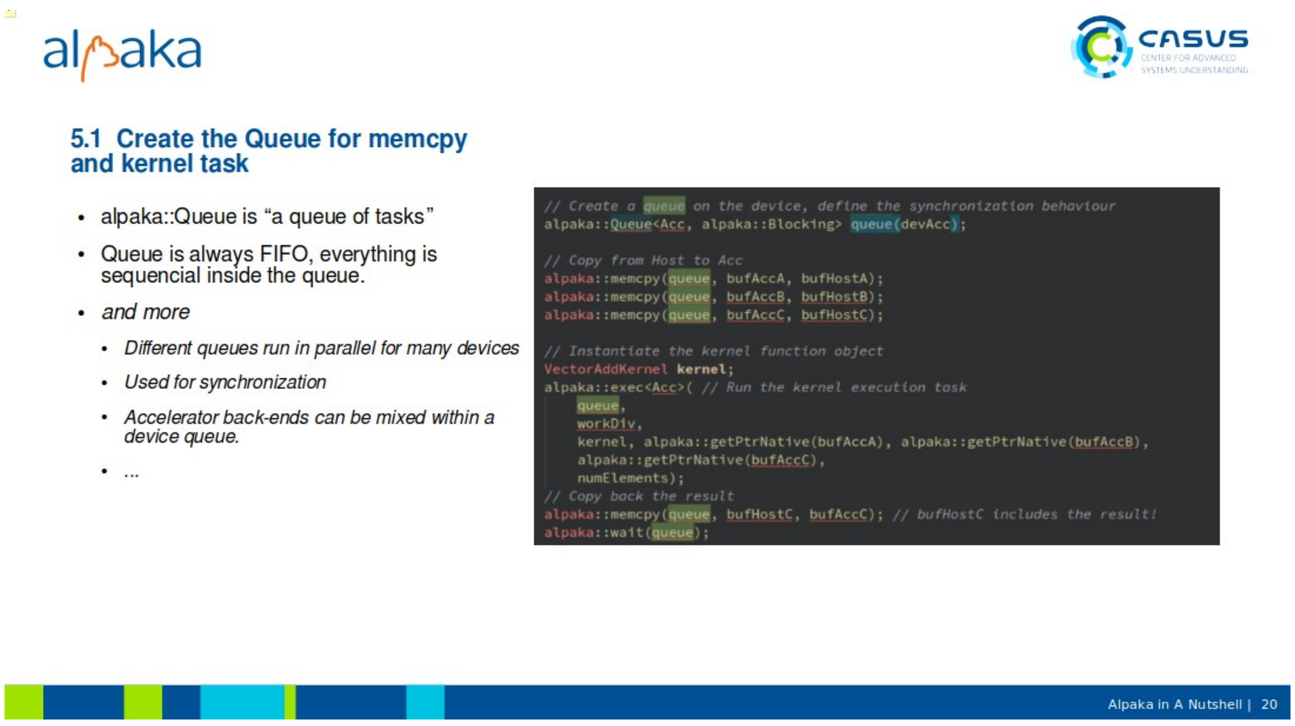
The forth step is Allocating memory for data vectors A and B on host and device.

For allocation of memory alpaka::Buf is used, it is multi-dimensional dynamic array.

It contains *memory adress*, *size*, the *device* it is the memory allocated!

alpaka::allocBuf() allocates memory to the given device.

alpaka::View is used to adapt existing memory, if we already have an STL vector filled with data at host for example we don't need to create Buf, getting a view of existing STL contiguous container would be ok.



alpaka

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

5.1 Create the Queue for memcpy and kernel task

- alpaka::Queue is “a queue of tasks”
- Queue is always FIFO, everything is sequencial inside the queue.
- *and more*
  - *Different queues run in parallel for many devices*
  - *Used for synchronization*
  - *Accelerator back-ends can be mixed within a device queue.*
  - ...

```
// Create a queue on the device, define the synchronization behaviour
alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

// Copy from Host to Acc
alpaka::memcpy(queue, bufAccA, bufHostA);
alpaka::memcpy(queue, bufAccB, bufHostB);
alpaka::memcpy(queue, bufAccC, bufHostC);

// Instantiate the kernel function object
VectorAddKernel kernel;
alpaka::exec<Acc>( // Run the kernel execution task
    queue,
    workDiv,
    kernel, alpaka::getPtrNative(bufAccA), alpaka::getPtrNative(bufAccB),
    alpaka::getPtrNative(bufAccC),
    numElements);
// Copy back the result
alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
alpaka::wait(queue);
```

Create **alpaka::Queue** using the device instance and the accelerator type (e.g GPU)

Queue is similar to stream in Cuda. Alpaka::Queue is always FIFO, everything is sequencial inside the queue.

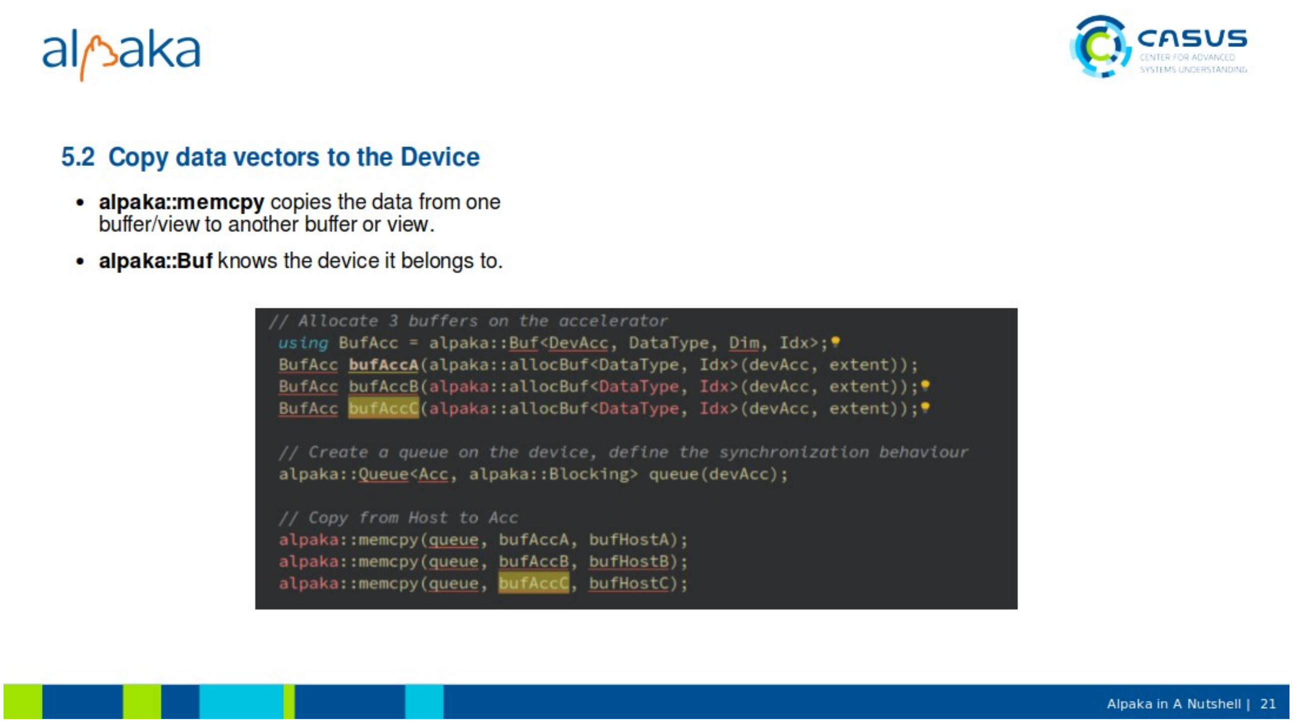
Two queue types: blocking and non-blocking.

Blocking means, the execution of task blocks the caller, in other words when a task is enqueued or executed; the calling thread is blocked. This property of does not affect relation between queues.

// Blocking-queues block the caller(host) until Device-side command returns.

if we create a non blocking queue using a CudaGpu accelerator type and nvidia device and another non blocking queue using the HipGpu accerator type and AMD device; host could execute a task on the first device then without being blocked could execute a task on the second device.

Since we are using single queue the operations on queue is always sequential we don’t need to think about weather the calling thread is blocked or not



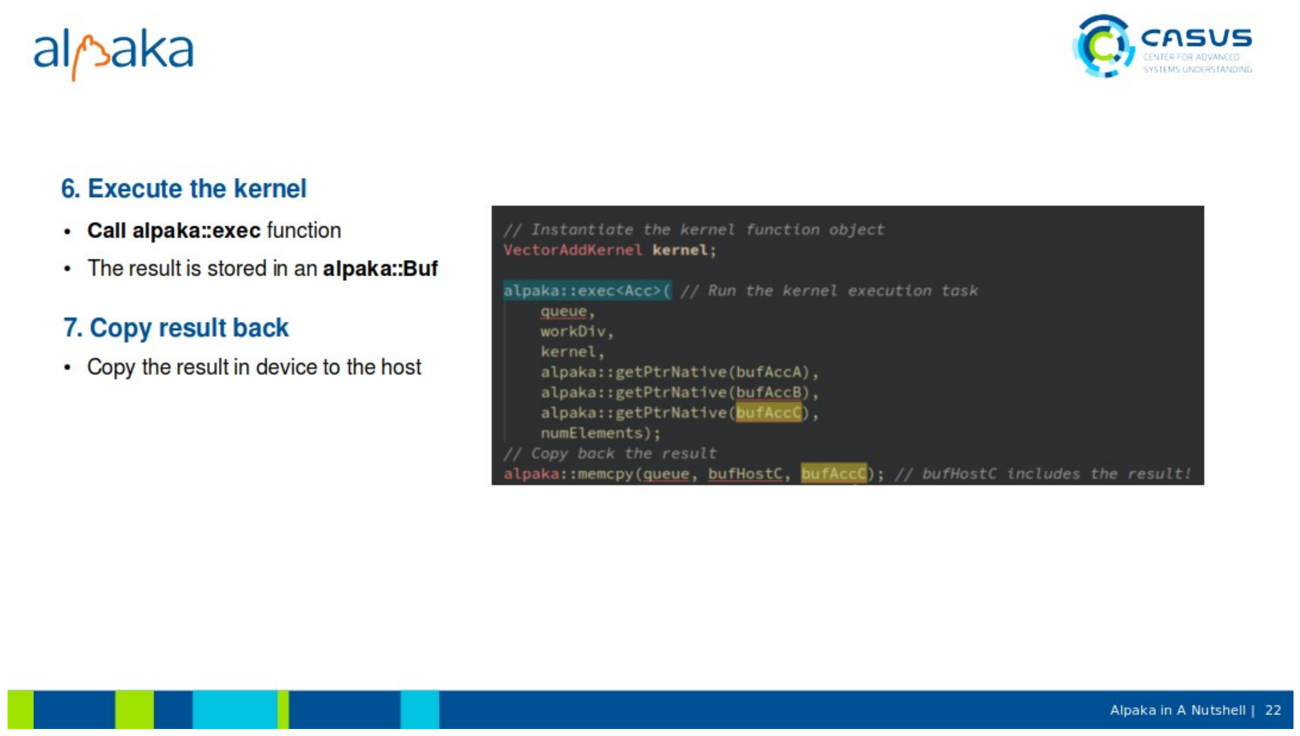
5.2 Copy data vectors to the Device

- **alpaka::memcpy** copies the data from one buffer/view to another buffer or view.
- **alpaka::Buf** knows the device it belongs to.

```
// Allocate 3 buffers on the accelerator
using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
BufAcc bufAccA(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
BufAcc bufAccB(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
BufAcc bufAccC(alpaka::allocBuf<DataType, Idx>(devAcc, extent));

// Create a queue on the device, define the synchronization behaviour
alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

// Copy from Host to Acc
alpaka::memcpy(queue, bufAccA, bufHostA);
alpaka::memcpy(queue, bufAccB, bufHostB);
alpaka::memcpy(queue, bufAccC, bufHostC);
```

because for all operations we used the same queue.

Copying data vectors to device done by alpaka::memcpy. Memcpy copies the second buffer argument to first buffer argument. As you would see in the highlighted memcpy call; only the buffers alllocated to host and device are given to function memcpy without stating at which device they are allocated. Because the device information is already inside the buffer data structure.


alpaka

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

6. Execute the kernel

- **Call alpaka::exec** function
- The result is stored in an **alpaka::Buf**

7. Copy result back

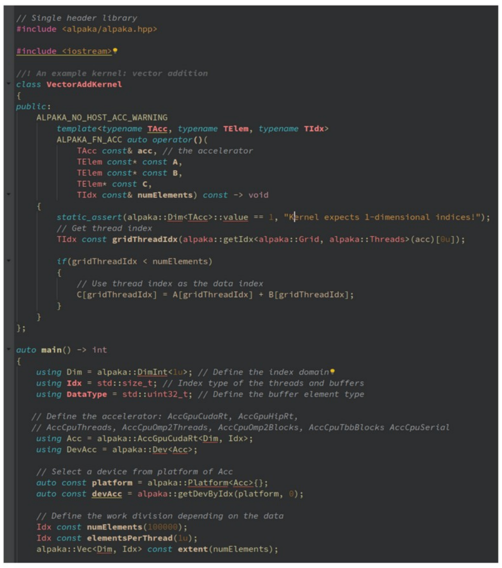
- Copy the result in device to the host

```
// Instantiate the kernel function object
VectorAddKernel kernel;

alpaka::exec<Acc>( // Run the kernel execution task
    queue,
    workDiv,
    kernel,
    alpaka::getPtrNative(bufAccA),
    alpaka::getPtrNative(bufAccB),
    alpaka::getPtrNative(bufAccC),
    numElements);
// Copy back the result
alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
```

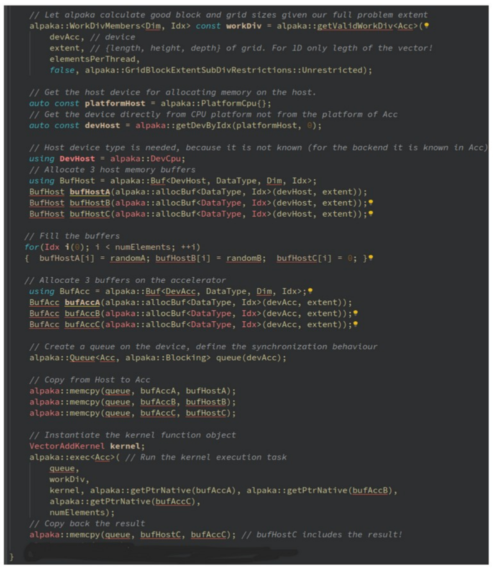
Alpaka in A Nutshell | 22

On the last 2 steps kernel is executed using the queue and result is copied back to the host. Since everything in the queue is sequential without being blocking and we used the same queue for 2 operations we are sure that there is no problem synchronization.

Parallel vector addition code

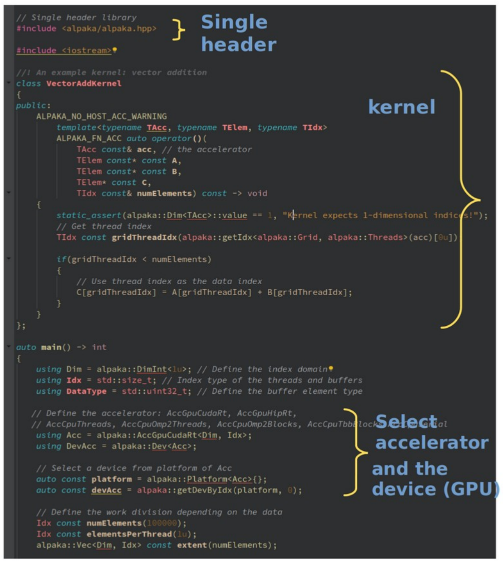


```cpp
// Single header library
#include <alpaka/alpaka.hpp>

#include <iostream>

//! An example kernel: vector addition
class VectorAddKernel
{
public:
    ALPAKA_NO_HOST_ACC_WARNING
        template<typename TAcc, typename TElem, typename TIdx>
        ALPAKA_FN_ACC auto operator()(
            TAcc const& acc, // the accelerator
            TElem const* const A,
            TElem const* const B,
            TElem* const C,
            TIdx const& numElements) const -> void
    {
        static_assert(alpaka::Dim<TAcc>::value == 1, "Kernel expects 1-dimensional indices!");
        // Get thread index
        TIdx const gridThreadIdx(alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u]);

        if(gridThreadIdx < numElements)
        {
            // Use thread index as the data index
            C[gridThreadIdx] = A[gridThreadIdx] + B[gridThreadIdx];
        }
    }
};

auto main() -> int
{
    using Dim = alpaka::DimInt<1u>; // Define the index domain
    using Idx = std::size_t; // Index type of the threads and buffers
    using DataType = std::uint32_t; // Define the buffer element type

   // Define the accelerator: AccGpuCudaRt, AccGpuHipRt,
   // AccCpuThreads, AccCpuOmp2Threads, AccCpuOmp2Blocks, AccCpuTbbBlocks AccCpuSerial
    using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
    using DevAcc = alpaka::Dev<Acc>;

    // Select a device from platform of Acc
    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);

    // Define the work division depending on the data
    Idx const numElements(100000);
    Idx const elementsPerThread(1u);
    alpaka::Vec<Dim, Idx> const extent(numElements);
```

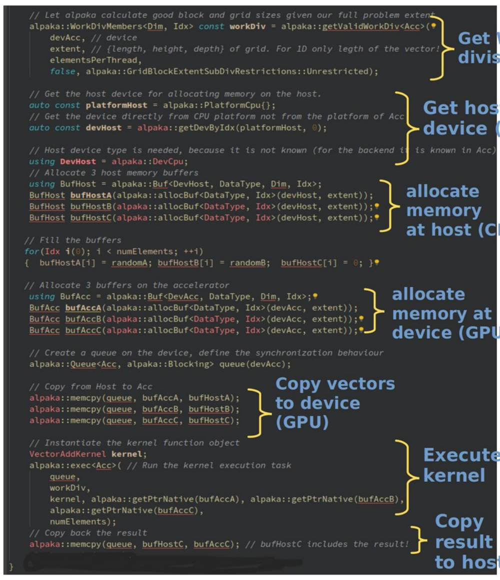


```cpp
    // Let alpaka calculate good block and grid sizes given our full problem extent
    alpaka::WorkDivMembers<Dim, Idx> const workDiv = alpaka::getValidWorkDiv<Acc>(
        devAcc, // device
        extent, // {length, height, depth} of grid. For 1D only legth of the vector!
        elementsPerThread,
        false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

    // Get the host device for allocating memory on the host.
    auto const platformHost = alpaka::PlatformCpu{};
    // Get the device directly from CPU platform not from the platform of Acc
    auto const devHost = alpaka::getDevByIdx(platformHost, 0);

    // Host device type is needed, because it is not known (for the backend it is known in Acc)
    using DevHost = alpaka::DevCpu;
    // Allocate 3 host memory buffers
    using BufHost = alpaka::Buf<DevHost, DataType, Dim, Idx>;
    BufHost bufHostA(alpaka::allocBuf<DataType, Idx>(devHost, extent));
    BufHost bufHostB(alpaka::allocBuf<DataType, Idx>(devHost, extent));
    BufHost bufHostC(alpaka::allocBuf<DataType, Idx>(devHost, extent));

   // Fill the buffers
   for(Idx i(0); i < numElements; ++i)
   {  bufHostA[i] = randomA; bufHostB[i] = randomB;  bufHostC[i] = 0; }

   // Allocate 3 buffers on the accelerator
    using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
    BufAcc bufAccA(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
    BufAcc bufAccB(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
    BufAcc bufAccC(alpaka::allocBuf<DataType, Idx>(devAcc, extent));

    // Create a queue on the device, define the synchronization behaviour
    alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

    // Copy from Host to Acc
    alpaka::memcpy(queue, bufAccA, bufHostA);
    alpaka::memcpy(queue, bufAccB, bufHostB);
    alpaka::memcpy(queue, bufAccC, bufHostC);

    // Instantiate the kernel function object
    VectorAddKernel kernel;
    alpaka::exec<Acc>( // Run the kernel execution task
        queue,
        workDiv,
        kernel, alpaka::getPtrNative(bufAccA), alpaka::getPtrNative(bufAccB),
        alpaka::getPtrNative(bufAccC),
        numElements);
    // Copy back the result
    alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
    [redacted]
}
```

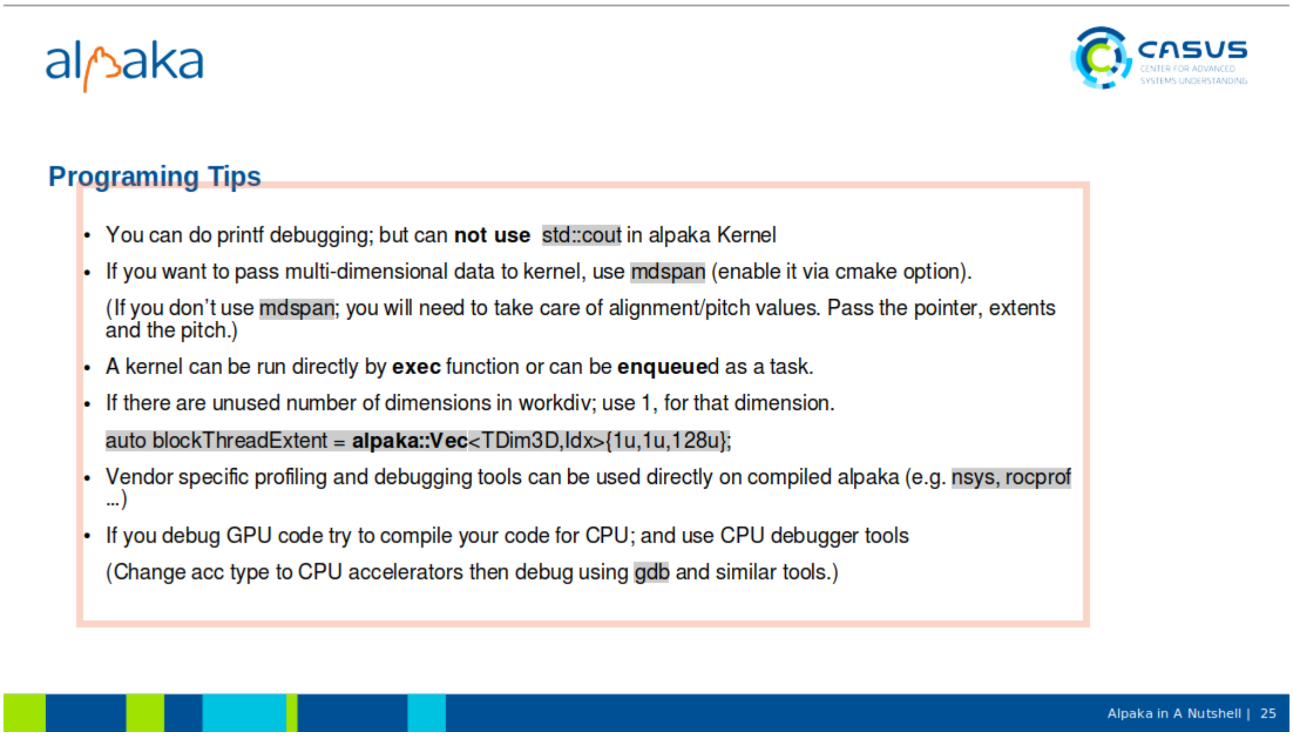
Parallel vector addition code



```cpp
// Single header library
#include <alpaka/alpaka.hpp>

#include <iostream>

//! An example kernel: vector addition
class VectorAddKernel
{
public:
    ALPAKA_NO_HOST_ACC_WARNING
        template<typename TAcc, typename TElem, typename TIdx>
        ALPAKA_FN_ACC auto operator()(
            TAcc const& acc, // the accelerator
            TElem const* const A,
            TElem const* const B,
            TElem* const C,
            TIdx const& numElements) const -> void
    {
        static_assert(alpaka::Dim<TAcc>::value == 1, "Kernel expects 1-dimensional indices!");
        // Get thread index
        TIdx const gridThreadIdx(alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u])

        if(gridThreadIdx < numElements)
        {
            // Use thread index as the data index
            C[gridThreadIdx] = A[gridThreadIdx] + B[gridThreadIdx];
        }
    }
};

auto main() -> int
{
    using Dim = alpaka::DimInt<1u>; // Define the index domain
    using Idx = std::size_t; // Index type of the threads and buffers
    using DataType = std::uint32_t; // Define the buffer element type

   // Define the accelerator: AccGpuCudaRt, AccGpuHipRt,
   // AccCpuThreads, AccCpuOmp2Threads, AccCpuOmp2Blocks, AccCpuTbbBlocks, AccCpuSerial
    using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
    using DevAcc = alpaka::Dev<Acc>;

    // Select a device from platform of Acc
    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);

    // Define the work division depending on the data
    Idx const numElements(100000);
    Idx const elementsPerThread(1u);
    alpaka::Vec<Dim, Idx> const extent(numElements);
```


Get Work division
Get host device (CPU)
allocate memory at host (CPU)
allocate memory at device (GPU)
Copy vectors to device (GPU)
Execute kernel
Copy result to host (CPU)

```cpp
    // Let alpaka calculate good block and grid sizes given our full problem extent
    alpaka::WorkDivMembers<Dim, Idx> const workDiv = alpaka::getValidWorkDiv<Acc>(
        devAcc, // device
        extent, // {length, height, depth} of grid. For 1D only legth of the vector!
        elementsPerThread,
        false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

    // Get the host device for allocating memory on the host.
    auto const platformHost = alpaka::PlatformCpu{};
    // Get the device directly from CPU platform not from the platform of Acc
    auto const devHost = alpaka::getDevByIdx(platformHost, 0);

    // Host device type is needed, because it is not known (for the backend it is known in Acc)
    using DevHost = alpaka::DevCpu;
    // Allocate 3 host memory buffers
    using BufHost = alpaka::Buf<DevHost, DataType, Dim, Idx>;
    BufHost bufHostA(alpaka::allocBuf<DataType, Idx>(devHost, extent));
    BufHost bufHostB(alpaka::allocBuf<DataType, Idx>(devHost, extent));
    BufHost bufHostC(alpaka::allocBuf<DataType, Idx>(devHost, extent));

   // Fill the buffers
   for(Idx i(0); i < numElements; ++i)
   {  bufHostA[i] = randomA; bufHostB[i] = randomB;  bufHostC[i] = 0; }

   // Allocate 3 buffers on the accelerator
    using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
    BufAcc bufAccA(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
    BufAcc bufAccB(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
    BufAcc bufAccC(alpaka::allocBuf<DataType, Idx>(devAcc, extent));

    // Create a queue on the device, define the synchronization behaviour
    alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

    // Copy from Host to Acc
    alpaka::memcpy(queue, bufAccA, bufHostA);
    alpaka::memcpy(queue, bufAccB, bufHostB);
    alpaka::memcpy(queue, bufAccC, bufHostC);

    // Instantiate the kernel function object
    VectorAddKernel kernel;
    alpaka::exec<Acc>( // Run the kernel execution task
        queue,
        workDiv,
        kernel, alpaka::getPtrNative(bufAccA), alpaka::getPtrNative(bufAccB),
        alpaka::getPtrNative(bufAccC),
        numElements);
    // Copy back the result
    alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
    [redacted]
}
```



Programing Tips

- You can do printf debugging; but can **not use** std::cout in alpaka Kernel
- If you want to pass multi-dimensional data to kernel, use mdspan (enable it via cmake option).

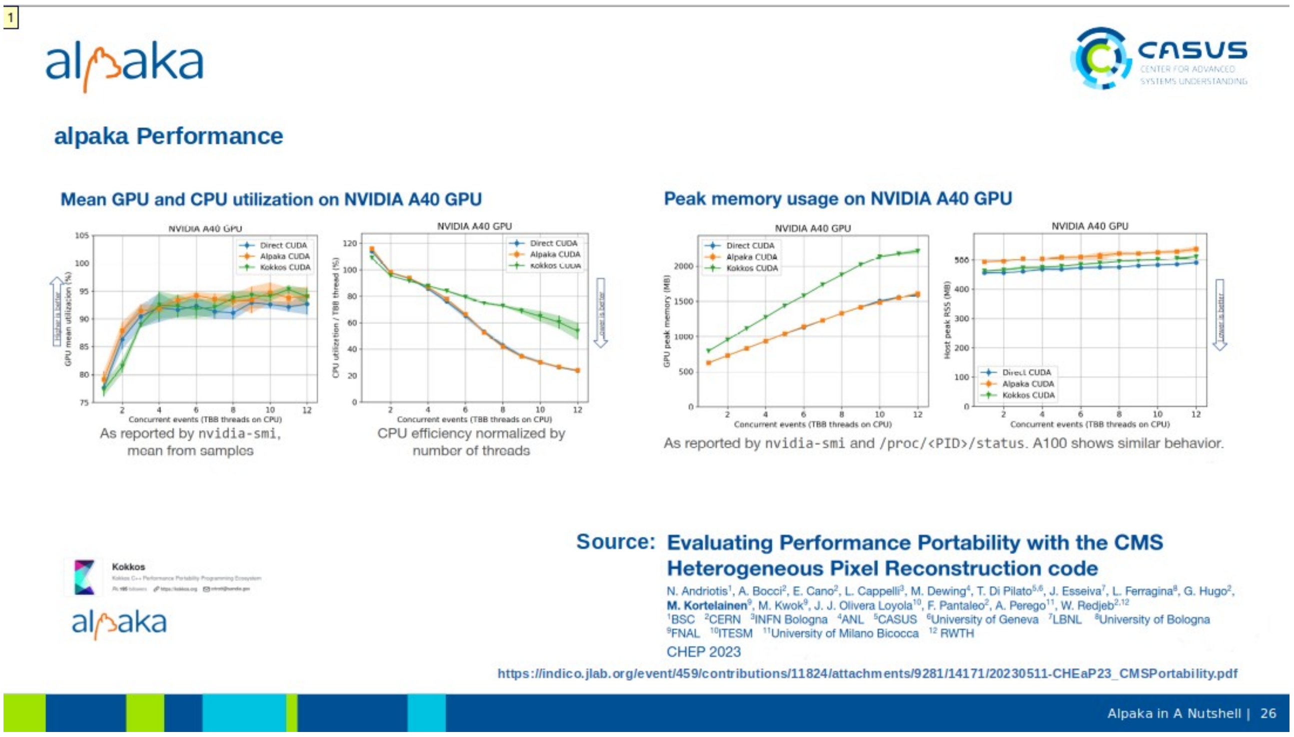
  (If you don't use mdspan; you will need to take care of alignment/pitch values. Pass the pointer, extents and the pitch.)
- A kernel can be run directly by **exec** function or can be **enqueue**d as a task.
- If there are unused number of dimensions in workdiv; use 1, for that dimension.

  auto blockThreadExtent = **alpaka::Vec**<TDim3D,Idx>{1u,1u,128u};
- Vendor specific profiling and debugging tools can be used directly on compiled alpaka (e.g. nsys, rocprof ...)
- If you debug GPU code try to compile your code for CPU; and use CPU debugger tools

  (Change acc type to CPU accelerators then debug using gdb and similar tools.)

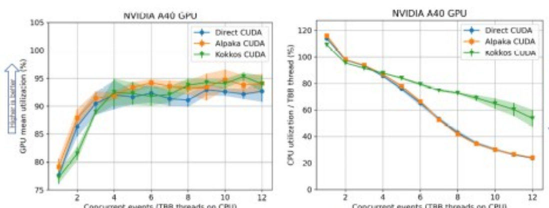
I'd like to give some programming tips for users of Alpaka.

1



alpaka Performance

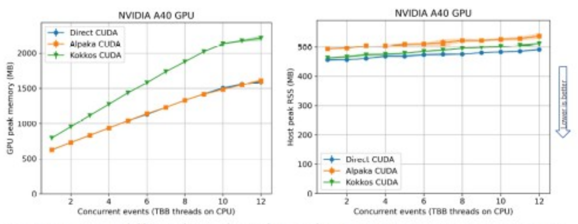
Mean GPU and CPU utilization on NVIDIA A40 GPU



As reported by nvidia-smi, mean from samples

CPU efficiency normalized by number of threads

Peak memory usage on NVIDIA A40 GPU



As reported by nvidia-smi and /proc/<PID>/status. A100 shows similar behavior.

Kokkos

Source: Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code

N. Andriotis[1], A. Bocci[2], E. Cano[2], L. Cappelli[3], M. Dewing[4], T. Di Pilato[5,6], J. Esseiva[7], L. Ferragina[8], G. Hugo[2], **M. Kortelainen**[9], M. Kwok[9], J. J. Olivera Loyola[10], F. Pantaleo[2], A. Perego[11], W. Redjeb[2,12]
[1]BSC [2]CERN [3]INFN Bologna [4]ANL [5]CASUS [6]University of Geneva [7]LBNL [8]University of Bologna [9]FNAL [10]ITESM [11]University of Milano Bicocca [12] RWTH
CHEP 2023

https://indico.jlab.org/event/459/contributions/11824/attachments/9281/14171/20230511-CHEaP23_CMSPortability.pdf

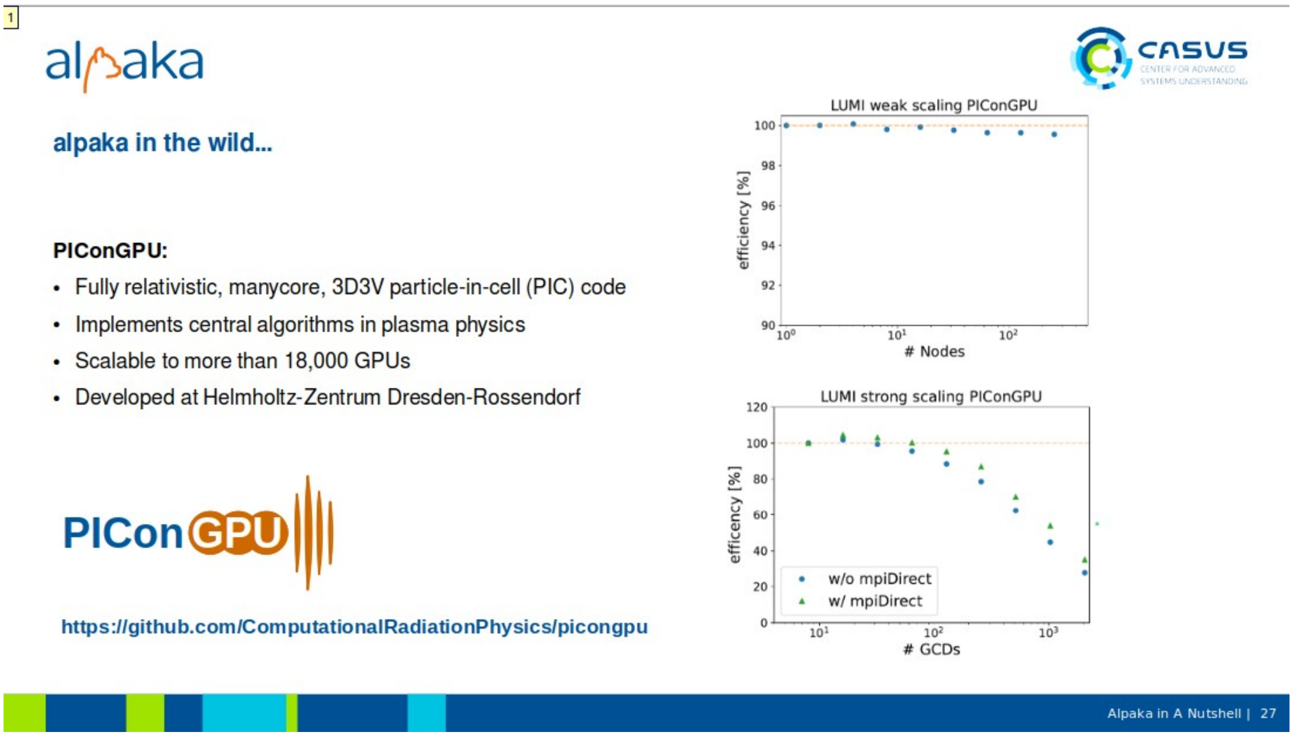
Alpaka in A Nutshell | 26

A study for understanding the performance of Alpaka was carried out by alpaka contributers and maintainers at CERN. 3 parallel programing tools are compared: Alpaka, a similar abstraction tool called Kokkos and Cuda itself.

The 2 graphs on the lefts show the GPU and CPU utilisation. In the case of mean GPU utilisation; performances of 3 tools were similar as you can see on the left most graph. But the CPU utilisation of alpaka was much better.

The 2 graphs on the right show the peak memory usage. And the memory usage of Alpaka was much better for the GPU and slightly better for the CPU compared to Kokos.

This performance analysis is done by CERN in 2021.



1

alpaka

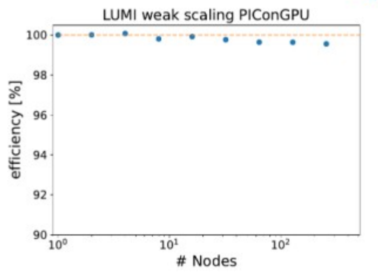
alpaka in the wild...

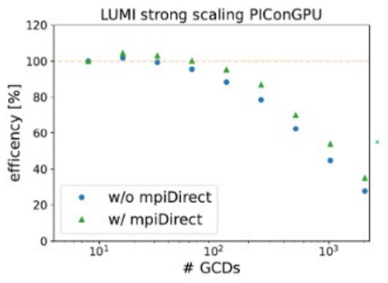
PIConGPU:

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

PIConGPU

https://github.com/ComputationalRadiationPhysics/picongpu





PiconGPU project is one of the most important users of alpaka.
It is "particle in cell "code and mainly does laser plasma acceleration.

It is run on some of the fastest super computers.

The graphs show how the performance of picongpu scales as the number of nodes grow according to tests performed on LUMI super computer. In the first graph the problem size increased with the number of size in the second one or the one below the problem size is fixed. OF course there is an underutilization of capacity.

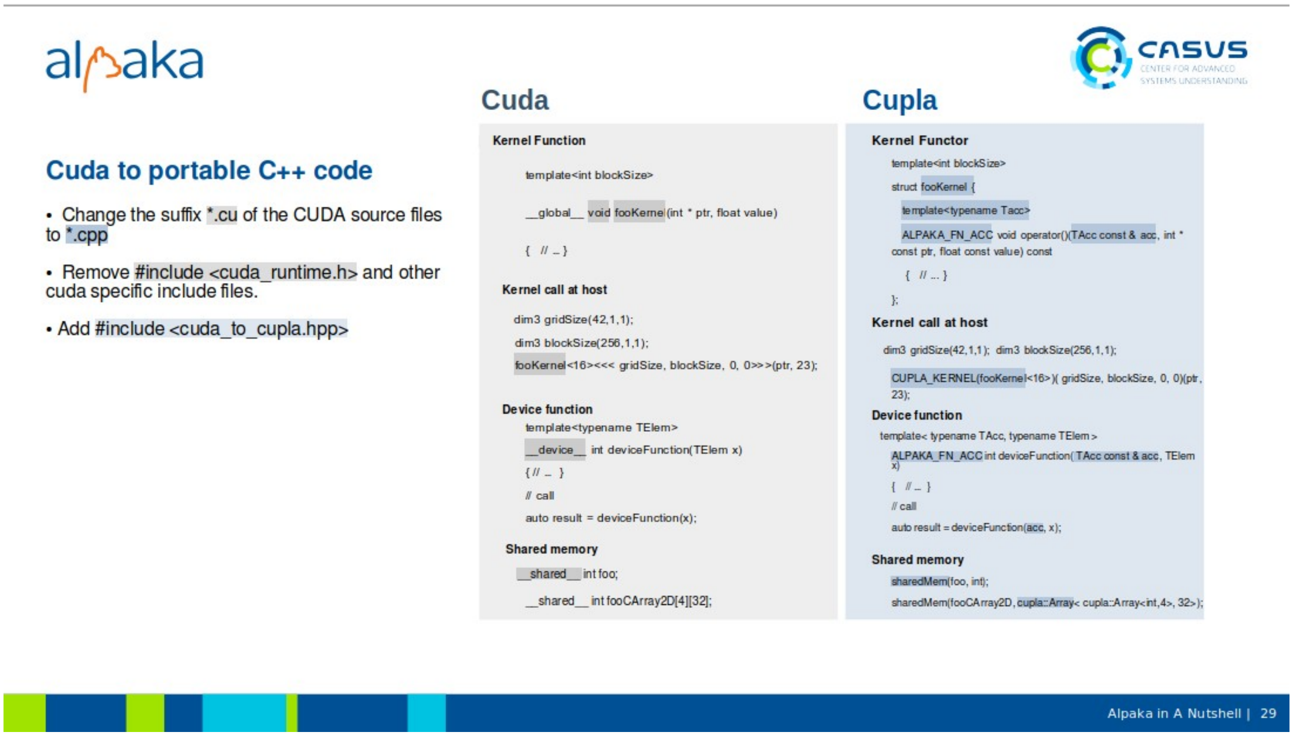
Hence Alpaka makes PicOnGPu to perform well on various different HPC platforms.

How to start using alpaka

- Don’t write code initially on **cuda** because **alpaka** is already low level!
- **Use alpaka directly** by using examples and the cheat-sheet.
- **BUT if you already have a codebase in cuda**, converting to **cupla** can be a fast solution to benefit from alpaka features! Cupla is a member of alpaka group of softwares.

cupla - C++ User interface for the Platform Independent Library [illegible]

https://github.com/alpaka-group/cupla



Cuda to portable C++ code

- Change the suffix *.cu of the CUDA source files to *.cpp
- Remove #include <cuda_runtime.h> and other cuda specific include files.
- Add #include <cuda_to_cupla.hpp>

Cuda

Kernel Function

```
template<int blockSize>

__global__ void fooKernel(int * ptr, float value)

{  // … }
```

Kernel call at host

```
dim3 gridSize(42,1,1);
dim3 blockSize(256,1,1);
fooKernel<16><<< gridSize, blockSize, 0, 0>>>(ptr, 23);
```

Device function

```
template<typename TElem>
__device__ int deviceFunction(TElem x)
{ // … }
// call
auto result = deviceFunction(x);
```

Shared memory

```
__shared__ int foo;
__shared__ int fooCArray2D[4][32];
```

Cupla

Kernel Functor

```
template<int blockSize>
struct fooKernel {
  template<typename Tacc>
  ALPAKA_FN_ACC void operator()(TAcc const & acc, int *
const ptr, float const value) const
   {  // ... }
};
```

Kernel call at host

```
dim3 gridSize(42,1,1);  dim3 blockSize(256,1,1);
CUPLA_KERNEL(fooKernel<16>)( gridSize, blockSize, 0, 0)(ptr,
23);
```

Device function

```
template< typename TAcc, typename TElem >
ALPAKA_FN_ACC int deviceFunction( TAcc const & acc, TElem
x)
{  // … }
// call
auto result = deviceFunction(acc, x);
```

Shared memory

```
sharedMem(foo, int);
sharedMem(fooCArray2D, cupla::Array< cupla::Array<int,4>, 32>);
```

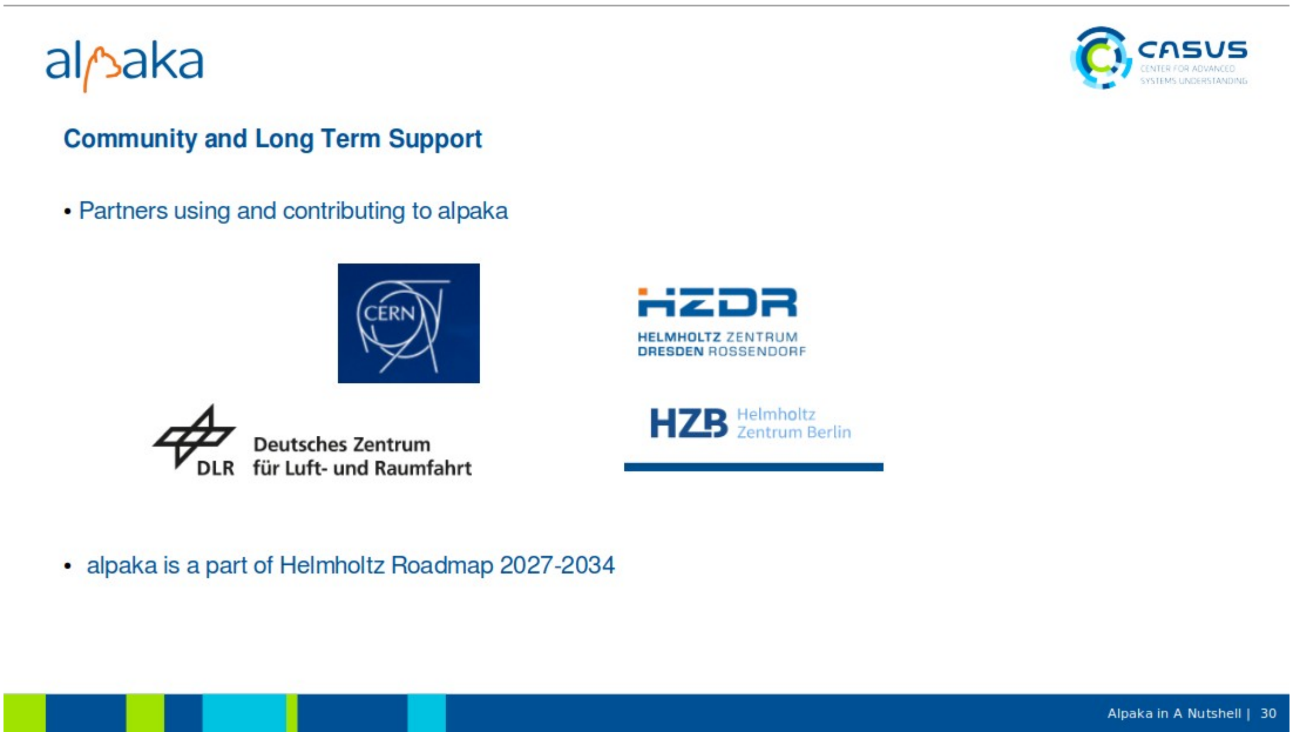
For converting Cuda to portable C++ code

Initially user needs to change the suffix *.cu of the CUDA source files to *.cpp

Remove the **#include <cuda_runtime.h> statement** and other include statements for the cuda specific include files.

And then the user has to include the file **cuda_to_cupla.hpp**

As you can see cuda and cupla code is quite similar. Cuda kernel function is converted to a kernel functor. Kernel call in cupla needs CUPLA_KERNEL macro and as an example shared memory declaration need sharedMem function. A 2d array in shared memory is is achieved my a cupla array of items of type cupla array.



Community and Long Term Support

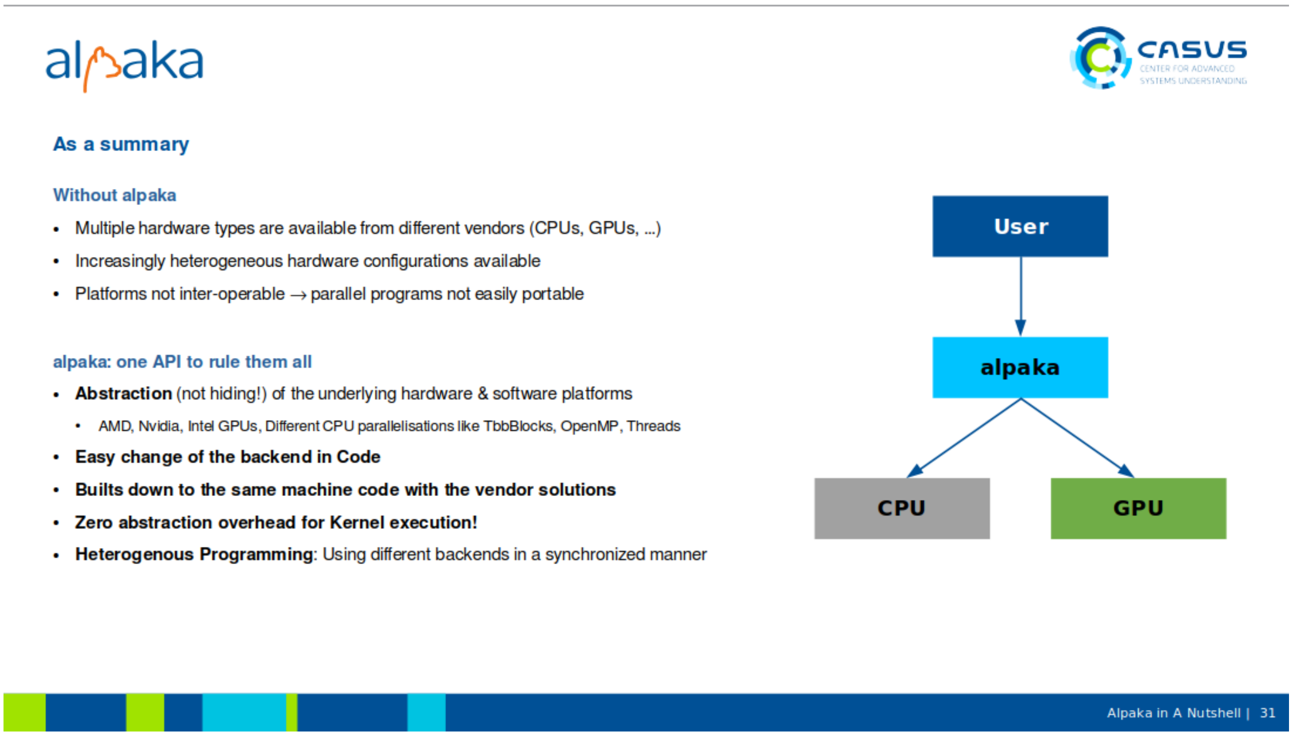
- Partners using and contributing to alpaka

- alpaka is a part of Helmholtz Roadmap 2027-2034

Before finishing I would like list the users of Alpaka library. CERN is a very important user and contributor, DLR is using alpaka but their codes are not on public domain. HZDR is an important user by creating PicOnGPU and HZDR is also directly contributing to develop and maintain alpaka. And lastly Helmholtz Zentrum Berlin has recently started using alpaka.

On the other hand since Alpaka is a part of Helmholtz Roadmap from 2027 to 2034, a long term support is already secured for Alpaka.

// examples select which backend
// CERN code links
// mdspan queue examples


As a summary

Without alpaka

- Multiple hardware types are available from different vendors (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, Different CPU parallelisations like TbbBlocks, OpenMP, Threads
- **Easy change of the backend in Code**
- **Builts down to the same machine code with the vendor solutions**
- **Zero abstraction overhead for Kernel execution!**
- **Heterogenous Programming**: Using different backends in a synchronized manner

As we mentioned before currently HPC Platforms are not interoperable, or in other words programs are not portable.

Alpaka provides one API to support all different GPUs and CPU beckends.

Alpaka provides Abstraction (but not hiding!) of the underlying hardware, compiler and OS

Alpaka does not have default device, built-in functions, language extentions, default stream like in cuda

It is Easy to change the backend in code

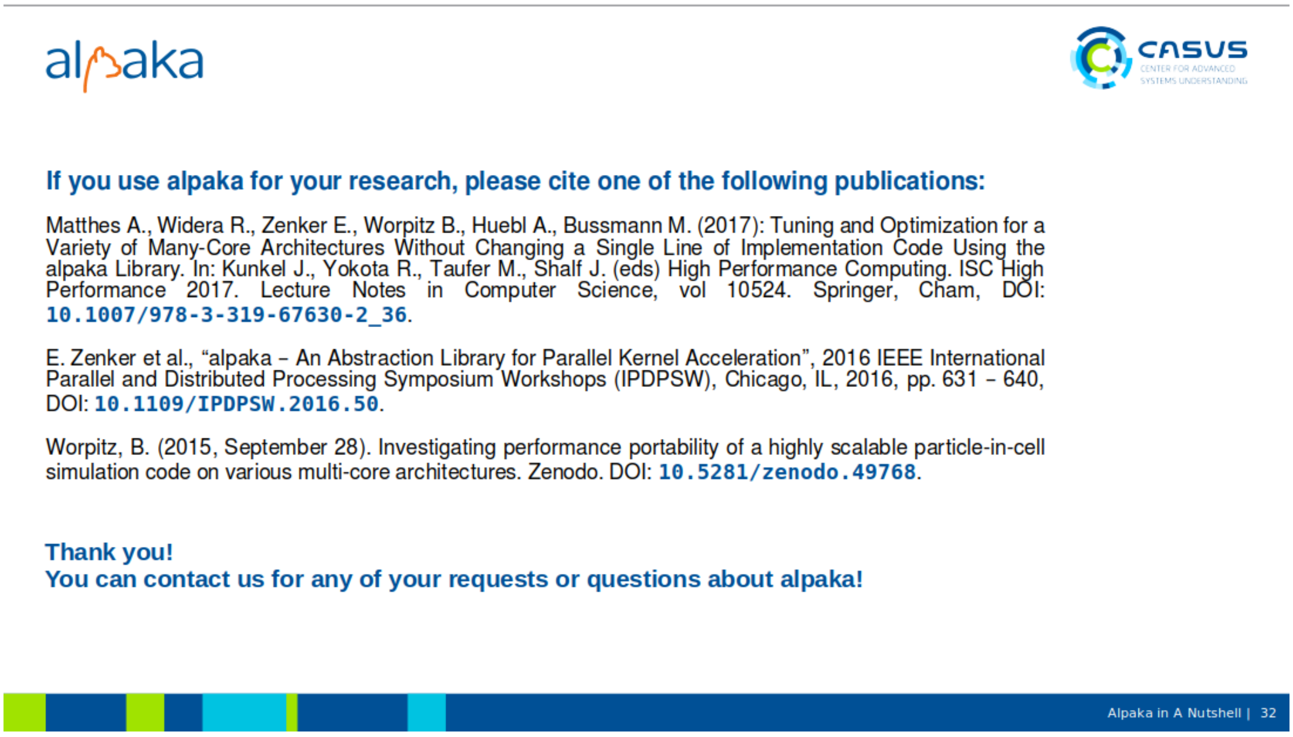
Alpaka code use directly of vendor APIs, does not depend on "unified APIs". Produces the same code that a vendor API would generate. It is not emulating. For example alpaka user can use vendor profilers and debuggers (Cuda,HIP...) for his alpaka code!

Supported GPU Backends are Hip (AMD), Cuda (NVidia), SYCL (Intel GPUs)

CPU Backends: OpenMp, Threads, TbbBlocks

Zero abstraction overhead for Kernel execution!

Heterogenous Programming: Using different backends in a synchronized manner



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`.

Thank you!
You can contact us for any of your requests or questions about alpaka!

If you use alpaka for your research please cite one of the publications.

Thank you for you attention. And Please feel free to contact us FOR any of your requests or questions about alpaka. It doesn’t matter if it is an installation issue, a bug or a performance problem.