alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

Plasma-PEPSC Webinar
28 May 2024

alpaka Parallel Programming Library

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

Slide1

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 give some more information about alpaka; like performance of alpaka, contributors and future plans.

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

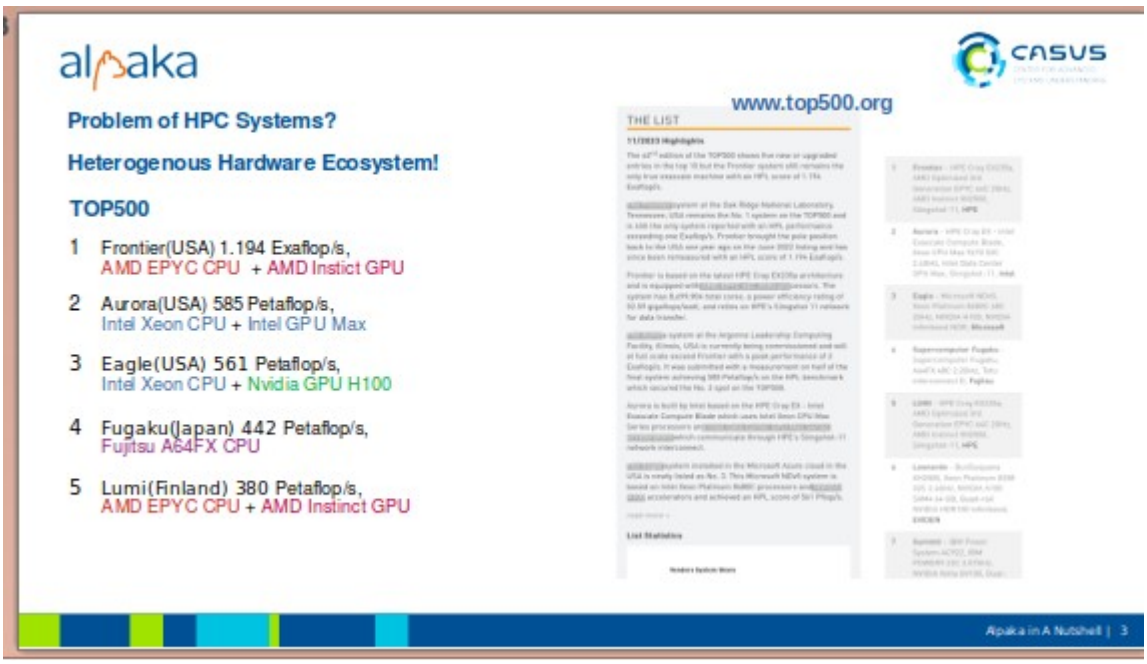
Slide 2

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 open-source and open to contributions.

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

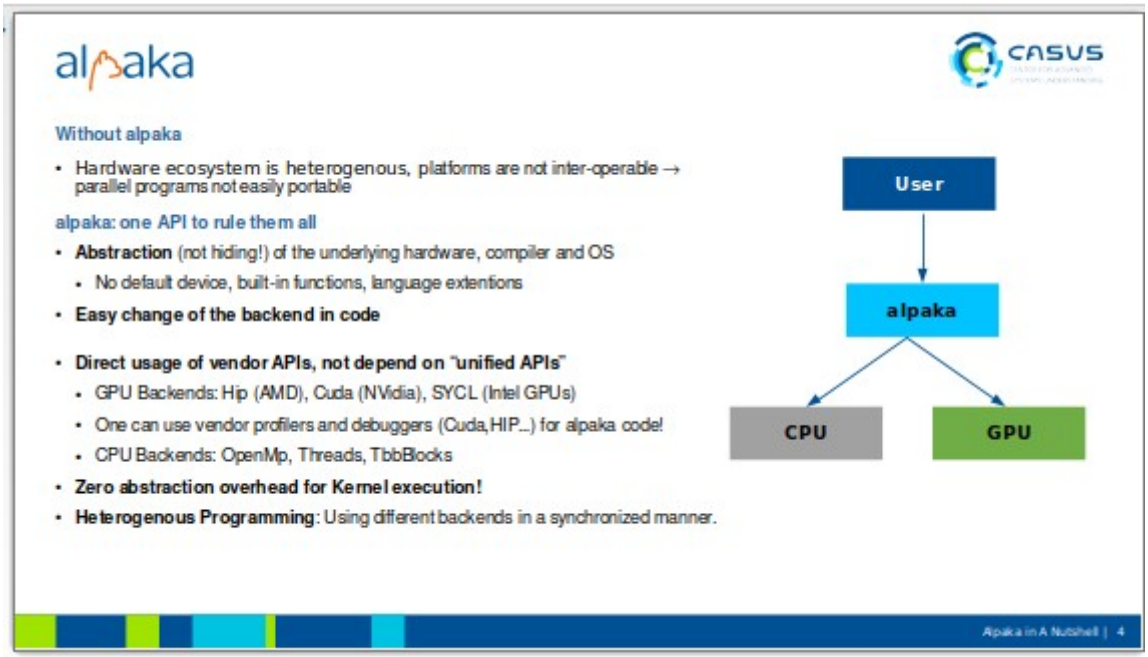


(Slide 3)

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. They are also using different colors not only in their logos but in their public appearance because of their brand building strategies.



(Slide4) 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 backends.

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

For example Alpaka does not have a default device, built-in functions, language extentions, or a default stream like in hip or cuda.

It is Easy to change “the backend” in the alpaka code

Alpaka code directly uses vendor 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 users can use vendor profilers and debuggers (Cuda,HIP…) for his alpaka code!

Supported CPU Backends of alpaka are OpenMp, Threads, TbbBlocks. Alpaka allows heterogenous Programming: Using

different backends in a synchronized manner is possible with alpaka.

(Slide 5) 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

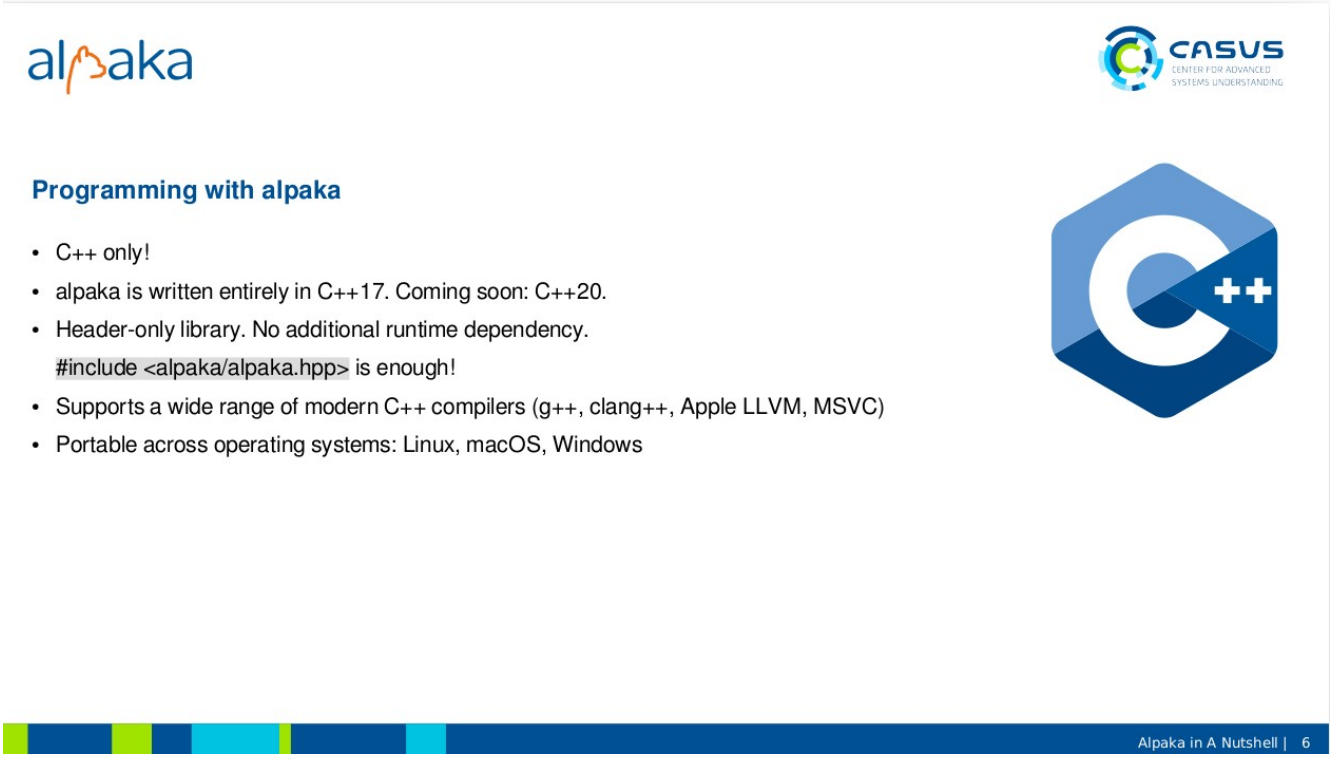
-Cheatsheet

-info about alpaka abstration model

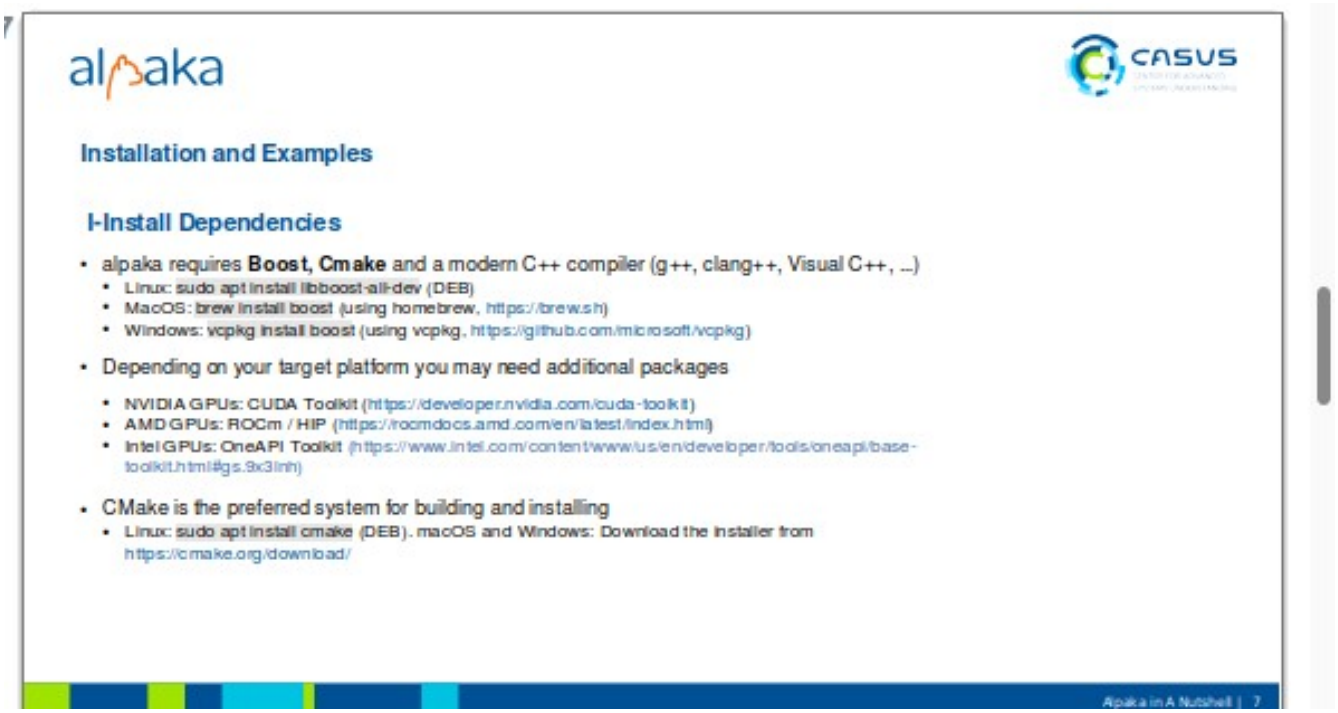
*** Alpaka Project group link at github contains all alpaka-related projects, documentation, samples, ...**

Among those softwares:
- cupla is an interface to alpaka for easy porting from cuda to a c++ code which uses alpaka

- vikunja is another alpaka group software. It is an API for alpaka for using high level algorithms like reduce or transform, because alpaka is low level.
alpaka is using (Mozilla Public License 2.0)



(Slide 6) Programming with Alpaka:
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. The only Compile time dependency is Boost.
Including the header file **alpaka.hpp** into the cpp code would be enough to use alpaka!
Alpaka Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MSVC)
and it is Portable across operating systems: Linux, macOS, Windows.
Actually every commit is tested an all compilers and operating system on the CI pipeline.



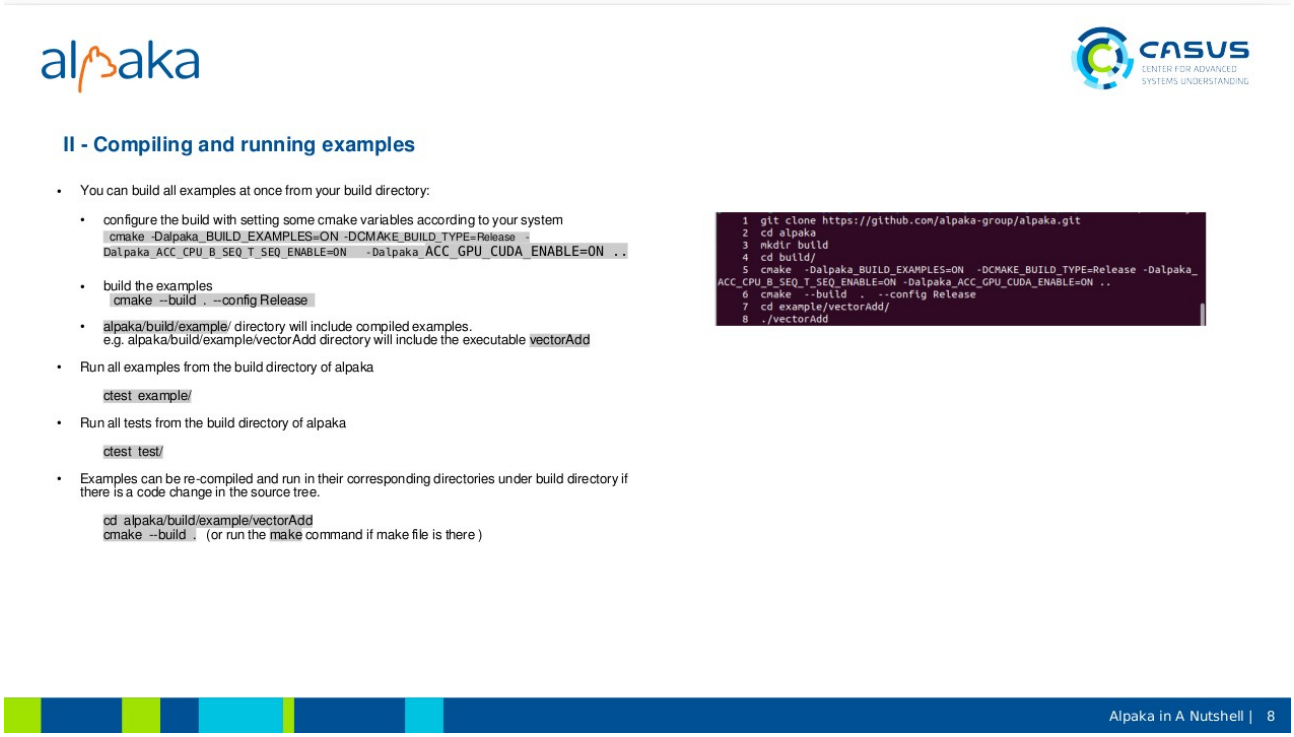
(Slide 7) Installation and Building Examples
I am going to quicly give some instructions on installation and building.

I- The first step is installing dependencies

alpaka requires Boost as a compile time dependency for compilation. For the configuration of the build system and compiling Cmake is needed although it is not a must.

Depending on your target platform you may need additional packages for example Cuda, Rocm or Intel OneAPI toolkits for different GPUs would be needed.

//CMake is the preferred system for building and installing



(Slide 8) After installing boost and cmake and compilation tools; you don't need to install alpaka files to compile the examples. You can just directly compile examples 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 using cmake --build, 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.

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

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

(Slide 9) Installation of alpaka is quite strait-forward. Compilation is not needed, just 2 steps are needed: configuration and installing.

Notice that you don’t need to set cmake variables depending on our system for installation, you can set set your CMAKE variables representing the available backends at your system at the executable level namely while compiling your examples or your project.

(Slide 10) Creating your alpaka project can be achieved in 2 ways. First way is, creating your **cmakelists.txt** 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 available accelerators while configuring the build tree by cmake. Your code will be portable between those selected backends.

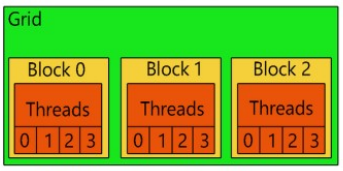
alpaka

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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.

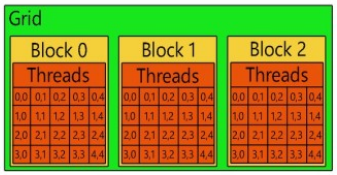


Grid-Block Extent: Vec{3}
Block-Thread Extent: Vec{4}

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

(slide 11) Before moving on a small alpaka code I would like to talk about basic concepts of thread-parallel programming.

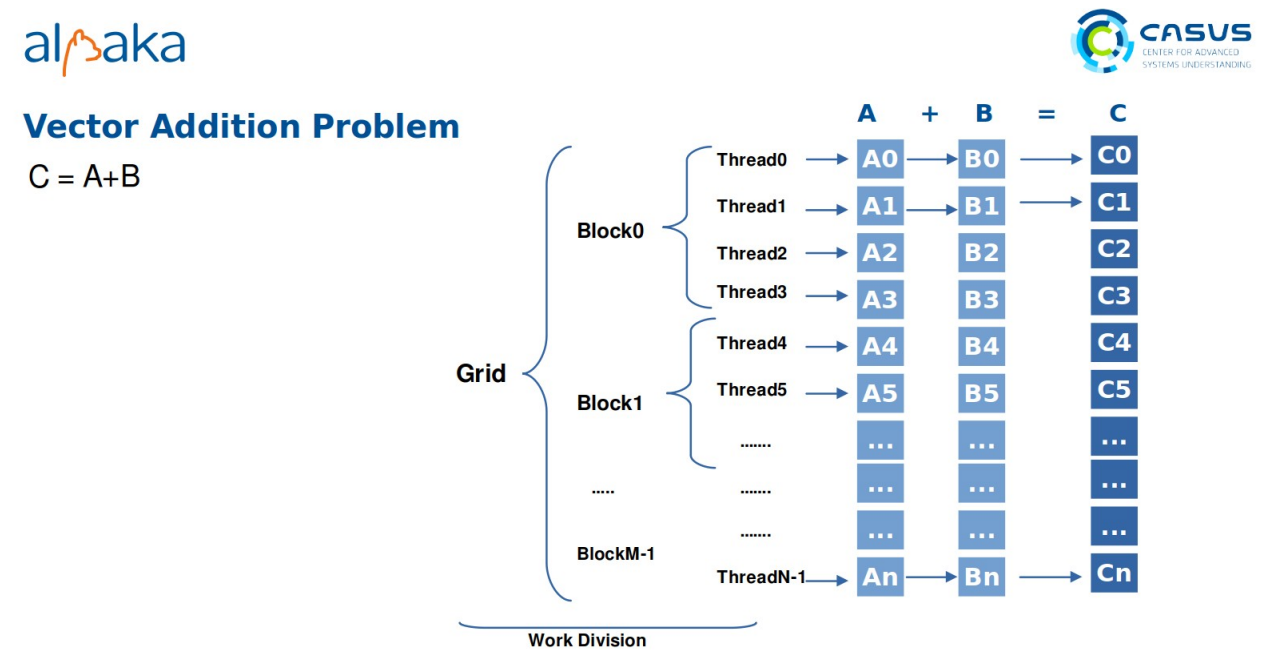
Alpaka uses Grid-block-thread based paralelisation model. The model is instantiated differently on different processors, because of cache size and the cache speed, the
synchronization mechanism, or simply the difference between CPU-GPUs. GPUs have many cores but small cache sizes on the other hand CPUs has many cores with large cache sizes. By using grid-block-thread abstraction the execution can be optimally adapted to the available hardware.

Secondly by thread-parallel programming it is meant that Large number of threads should run the same code (kernel) on different data in a parallel manner.

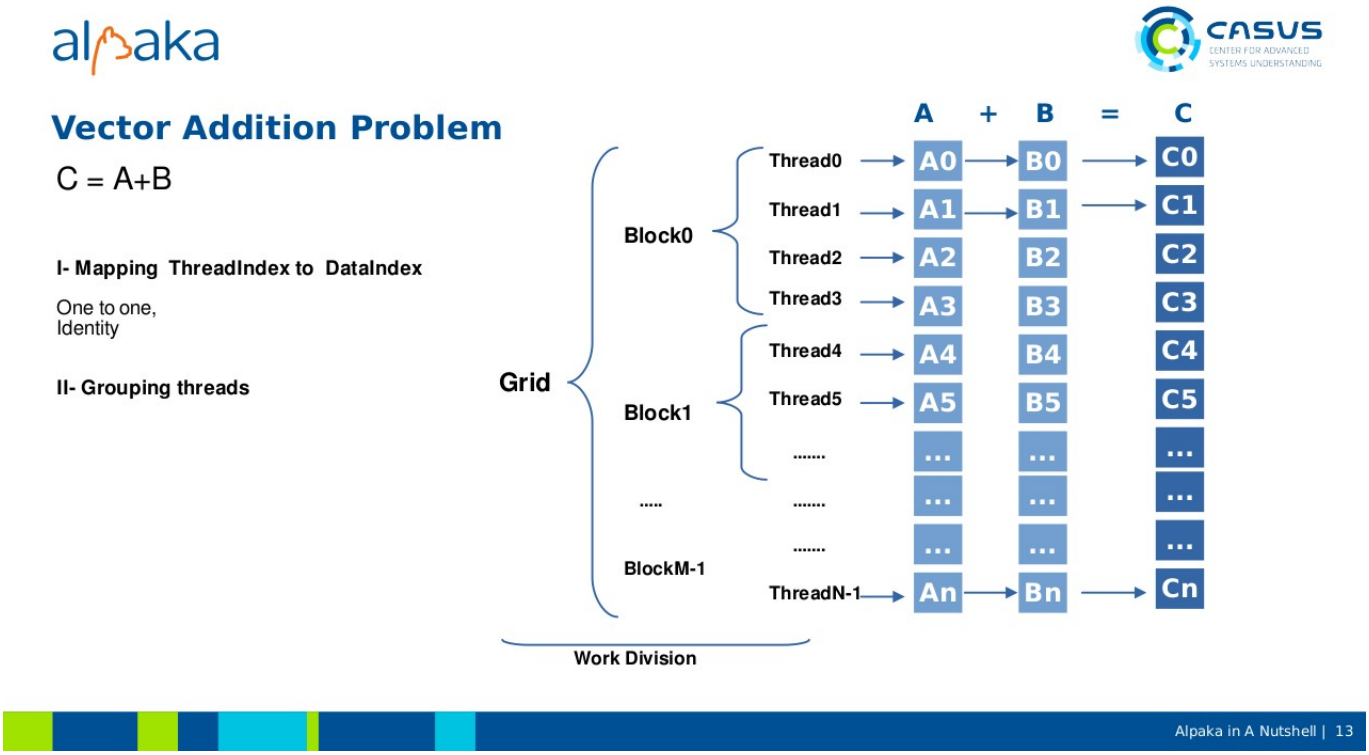
Lastly; Indexing of threads is needed. Each thread should work on a different data portion or do a specific task,
therefore each thread has an index accessible in kernel.

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 {depth,height and width}.
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. There are 2 grids on the slide.



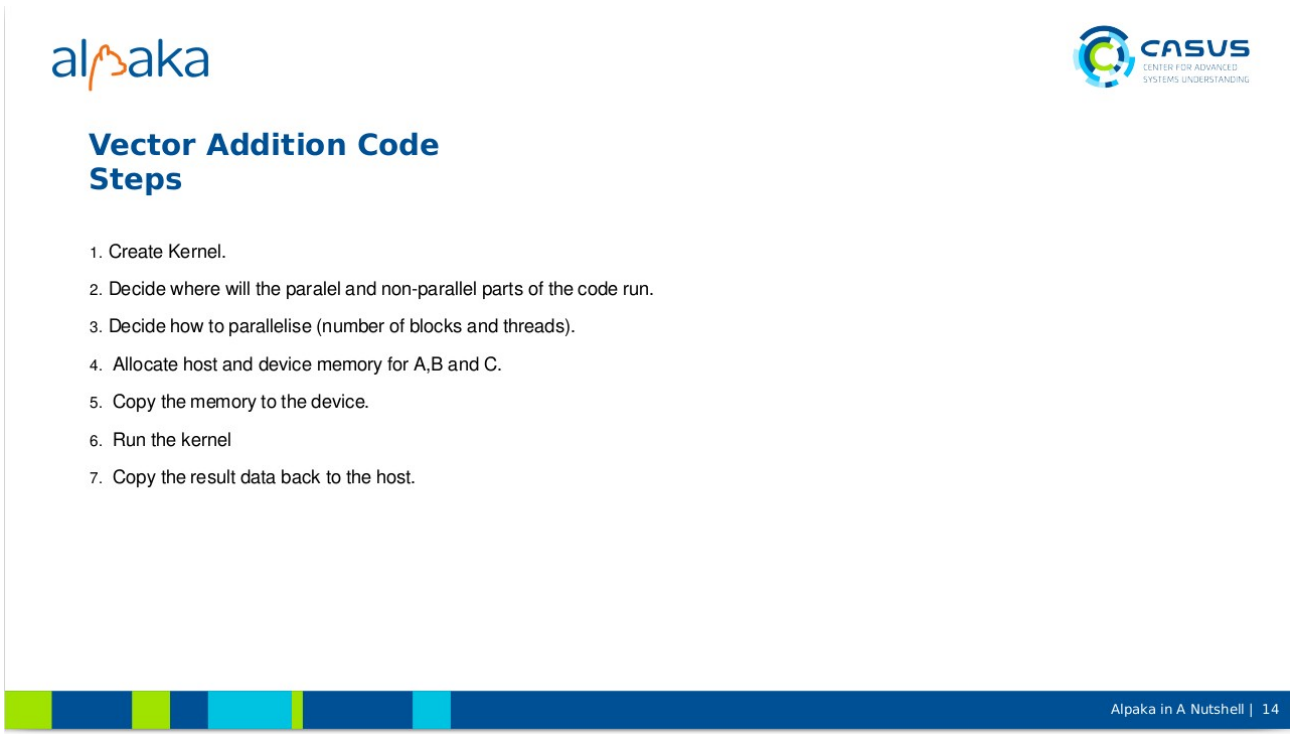
(Slide 12) Lets assume that we want to sum TWO vectors by utilising paralism of the hardware in a portable manner!!. We have TWO one dimensional vectors, vector A and vector B and we are going to calculate their sum C.



(Slide 13) 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. Grid block extent is M and block thread extent is selected 4 for the sake of the simpilicity of the drawing.



(Slide 14) Vector addition code steps are not difficult to guess.

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. Fill A and B with data.
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

```
// 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];
        }
    }
};
```


(Slide 15) At the first step kernel is defined. Kernel contains code that is run by each thread, and kernel knows the thread id which is instantiating the kernel itself. Alpaka kernels are functors, namely structs with specificly implemented function operators or lambdas.

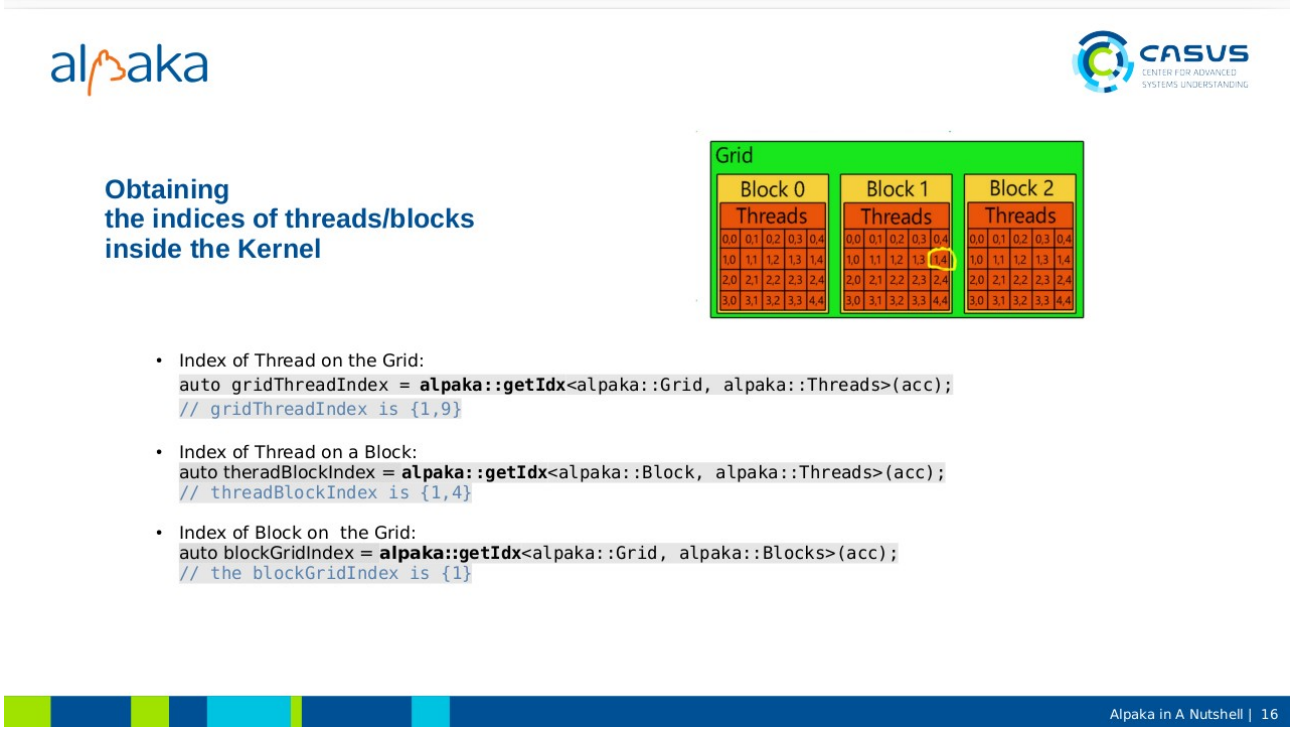
Arguments of the function operator can be pointers or trivially copyable types. (You can put many pointers and built-in types in a struct and just pass the struct as a value for example.)

`ALPAKA_FN_ACC` is required for kernels and functions called by device, `acc` is mandatory first parameter of the kernel, its type is the template parameter

Alpaka kernel is agnostic to device details which makes it generic actually.

As you see, Alpaka is low level and transparent. The term “Abstraction” usually associated with being high level but alpaka is

low level. One can access to the thread index directly as in Hip or Cuda.



(Slide 16) 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 10^{th} 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.

Block-Grid index for that specific thread will be 1 because it is the block with index 1.

(slide 17) Now we can design the code part that will NOT run in parallel, namely the code that will run on the host-device.

Initialy selecting one of the predefined accelerator types (or backends) according to your system is needed. For systems using GPUs there are 3 accelerator types, for the CPU backend there are 4 and for the Fpga backend there is an accelerator type defined.

In this example AccGpuCudaRt namely GPU Cuda backend is selected. But You can easily change the accelerator type in the code and then the code would run the kernel ON another backend.

Accelerator is a type that is only instantiated on Device (in kernel) not on the host.

An instance of device is needed on the host side because we are going to allocate buffers to the GPU for example and copy data to that device before running the kernel. (//ask this sentence)

For easy management of physical devices alpaka has a device data structure.

(slide 18) “How to paralelise” means How many blocks will be in the grid and how many threads each block will have?

There are 2 ways in alpaka: 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 this box of threads into thread blocks.

Elements per thread extent is a sub indexing which could be assigned to each thread; if each thread needs a number of additional sub indices set or an extent.

The generated block size by getValidWorkDiv will be inside the allowed limits which is usually 1024 threads per block for GPUs. (// what other limits we have?) There are additional inputs of **GetValidWorkDiv** function which are not used in this example; but you can ask to the get validworkdiv function “Generate blocks with EqualExtent”, then the extent dimensions would be equal.

Secondly work division can be just defined by the user. In that case 3 vectors; Grid-block, block thread extent and thread-elem extent should be determined by the user. You can determine it According to your experience or trial error. (// register spill ?)

// (There are vector registers in CPUs which can use vectorize basic operations, SIMD. Hence this is a similar abstraction to SIMD)

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

- **alpaka::Buf** is multi-dimensional dynamic (runtime sized) contiguous container.

  It contains
  - *memory*,
  - *size*,
  - the *device*, the memory is allocated in!
- **alpaka::allocBuf()** allocates memory to the given device.

```
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 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, still not known
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 host 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));
```

Alpaka in A Nutshell | 19

(slide 19) The forth step is Allocating memory for data vectors A and B on host and device.

For allocation of memory alpaka::Buf type is used, it is multi-dimensional dynamic (runtime sized) container.

It contains *memory adress*, the extent *of the data*, and the *device*! Since the buffer knows the device to which it belongs; it is easier to use in a lets say heterogenous environment.

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

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 sequential inside the queue.
- *and more*
  - *Different queues run in parallel for many devices*
  - *Used for synchronization*
  - *Accelerator back-ends can be mixed (used in interleaves) 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);
```


(slide 20) Create **alpaka::Queue** by using the device instance and the accelerator type (e.g GPU)

Queue is similar to the “stream” construct in Hip or Cuda. But Alpaka has no default queue. Alpaka::Queue is always FIFO, everything is sequential inside the queue. As you can see queue is used in memcpy and kernel execution, hence we are sure that everything will be in order.

Two queue types: blocking and non-blocking which is relevant if we want to have two queues.

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

if we create a non blocking queue using a CudaGpu accelerator type and an nvidia device instance; and another non blocking queue using the HipGpu accerator type and an AMD device instance; 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 because for all operations we used the same queue.

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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.
- Alternatively **alpaka::View** is used to adapt already allocated memory.

  if we already have a C++ **std::vector** at host; we don't need to create an **alpaka::Buf** to copy it between different devices. Converting it to an **alpaka::View** is enough.

```
// 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));

// Define the synchronization behavior of a queue
// choose between Blocking and NonBlocking
// Create a queue on the device
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);
```


(Slide 21) Copying data vectors to device done by alpaka::memcpy. Memcpy copies the data at the buffer passed as the second argument to buffer passed as the first 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 or the size. Because the device information is already inside the buffer data structure. (// how can we pass the buffer to memcopy is it an instance or a pointer)

alpaka::View is used to adapt existing memory, if we already have an STL vector filled with data at host for example or any other contiguous container; we don't need to create alpaka::Buf, getting a view of existing STL contiguous container would be ok. So if you say "I am used to using STL containers what I will do" you can use them till you need the copy data to the device.

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!
```


(slide 22) On the last 2 steps kernel is executed or actually enqueued in the queue; and result is copied back to the host. To execute kernel exec function is used. Queue, workdiv, kernel instance are first 3 arguments of exec; the rest are kernel arguments.

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 of 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!
}
```


(Slide 23)

The whole code looks like this!

Parallel vector addition code

(Slide 24)

alpaka

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Programing Tips

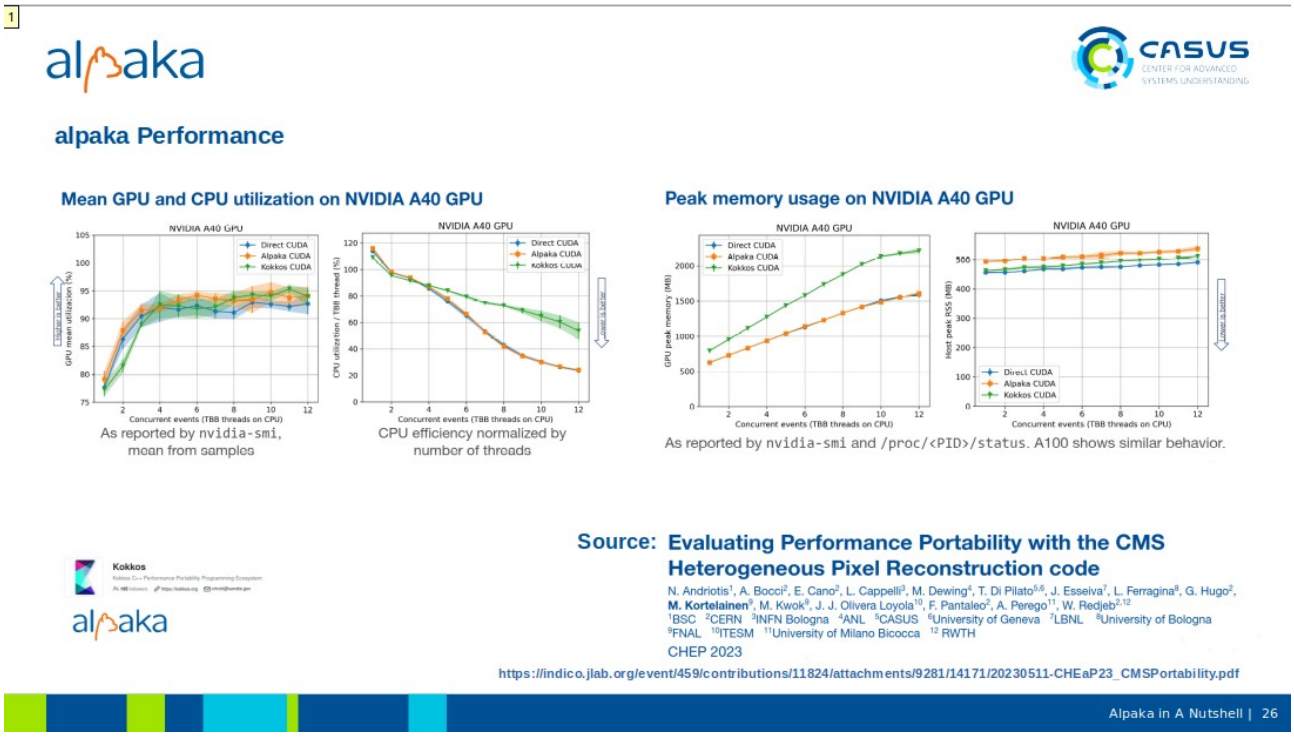
- If you want to pass multi-dimensional data to kernel, use `mdspan` (enable it via cmake option)

  (If you don't use `mdspan`; you will need to take care of alignment/pitch values. Pass the pointer, extents and the pitch.)
- To incease perfomance; using **shared memory and constant memory** of GPUs are among alpaka features.
- A kernel can be run directly by **exec** function or can be **enqueue**d as a task.
- Vendor specific profiling and debugging tools (e.g. **nsys, rocprof** ...) can be used on compiled alpaka code.
- If you debug GPU code try to compile your code for CPU; and use CPU debugger tools

  (Change accelerator type to CPU accelerators, then debug using **gdb** and similar tools.)
- Inside alpaka Kernel, you can use `printf`; but you should **not use** `std::cout` for GPU backends.
- If there are unused number of dimensions in workdiv; use 1, for that dimension.

```
auto blockThreadExtent = alpaka::Vec<TDim3D,Idx>{1u,1u,128u};
```


(Slide 25) I'd like to give some programming tips for users of Alpaka as the last slide of the programming part.

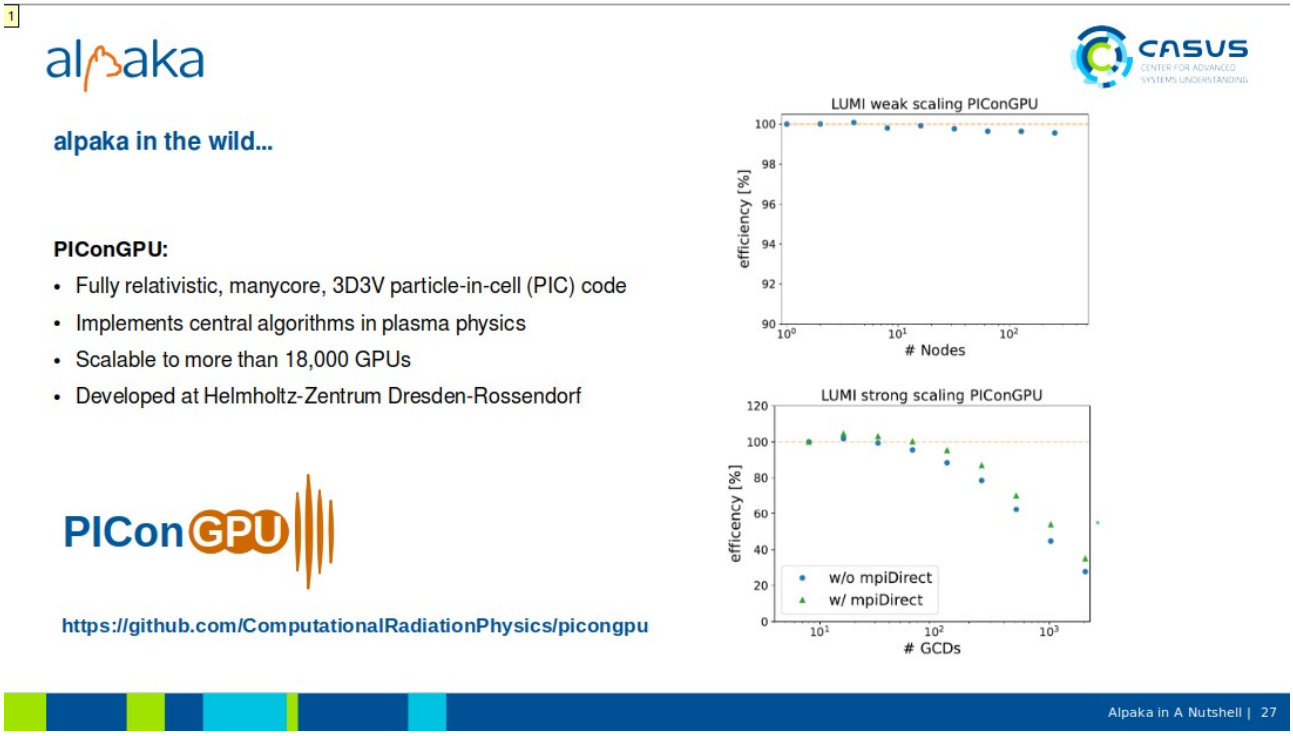


(Slide 26) performance of alpaka: A study for understanding the performance of Alpaka was carried out by alpaka contributers and maintainers at CERN. 3 parallel programing tools are compared: Alpaka, a similar abstraction tool called Kokkos and Cuda itself.

The 2 graphs on the left 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 against kokos and almost the same as cuda.

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 was done by CERN in 2021, an updated study is needed. We are planning to create benchmarks in a short time.



(Slide 27) PiconGPU project is one of the most important users of alpaka library.
It is “particle in cell“ code and mainly implements 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 Finland. 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. (// GCD : Graphics Compute Die. 2900 nodes each node has 4 AMD GPU modules and each has 2 dies)

Hence we can say that Alpaka makes PicOnGPu to perform well on various different HPC platforms.

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

How to start using alpaka

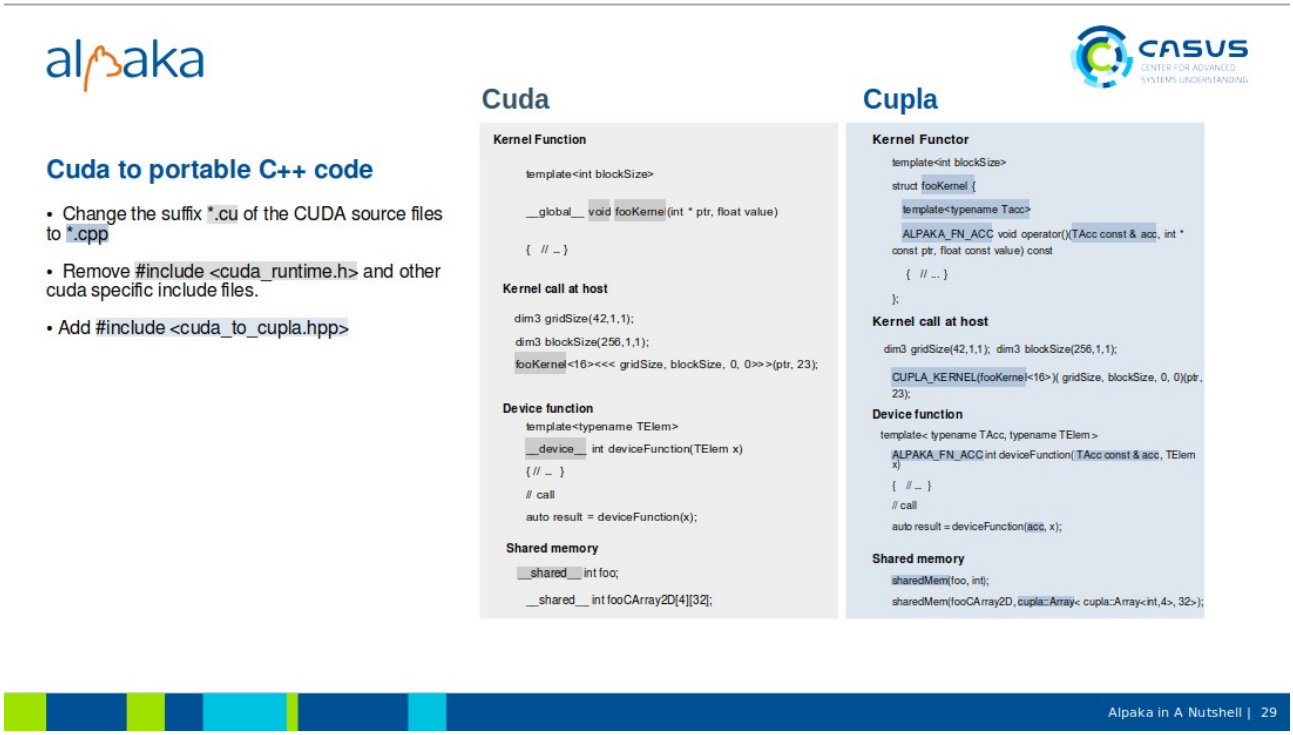
- Don't write code initially on **cuda** because **alpaka** is already low level!
- **Use alpaka directly** by using examples and the cheat-sheet.
- **BUT if you already have a codebase in cuda**, converting to **cupla** can be a fast solution to benefit from alpaka features! Cupla is a member of alpaka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

cupla - C++ User interface for the Platform Independent Library

cupla

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

(Slide28)



(Slide29) **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> directive** and other include directives 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. AS an example the shared memory declaration needs sharedMem function in CUPLA. A 2d array in shared memory is is achieved by a cupla array of cupla arrays.

(Slide 30) Before finishing I would like list the users and contributors 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 by 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 Strategic Helmholtz Program-Oriented funding Roadmap from 2027 to 2034, a long term support is already secured for Alpaka.

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

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

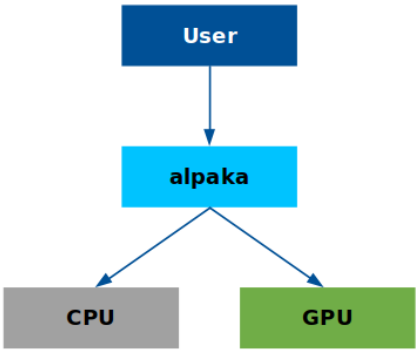
As a summary

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



(Slide31) 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

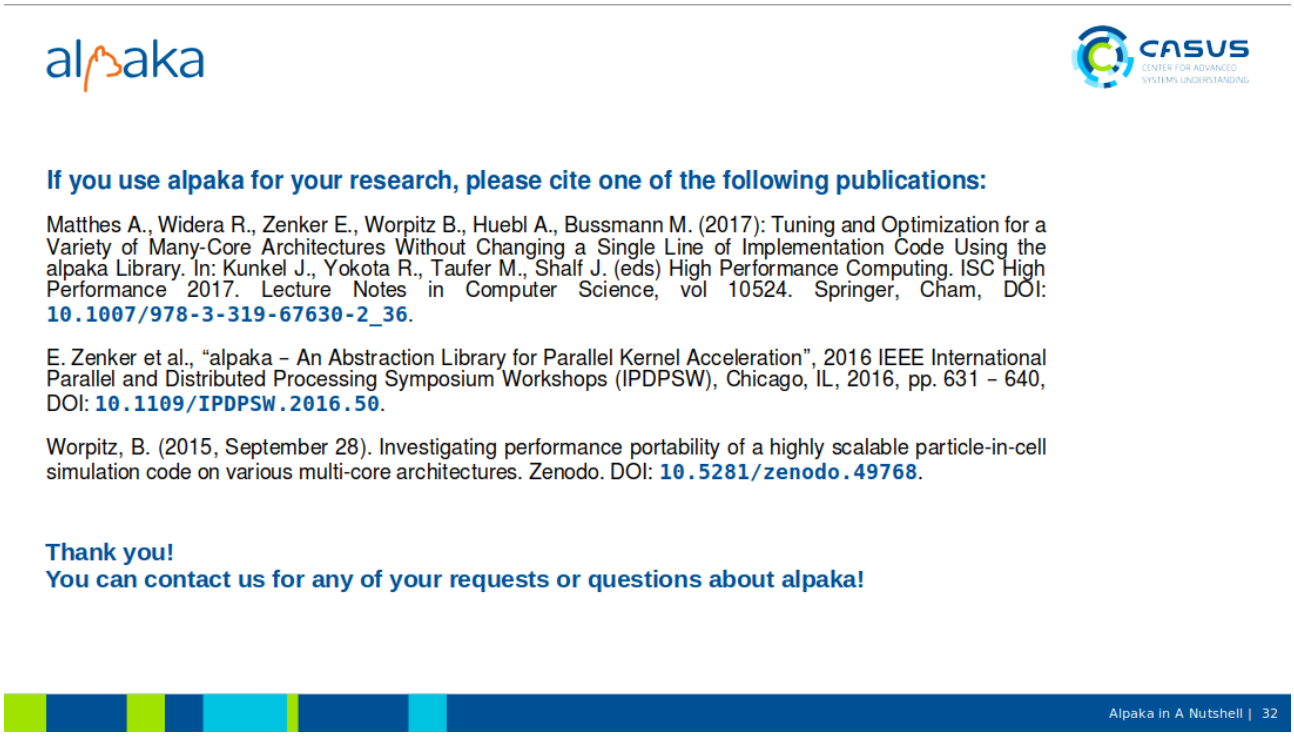
Alpaka does not have default device, built-in functions, language extentions, default stream like in cuda

It is Easy to change the backend in code

Alpaka code use directly of vendor APIs. 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!**

Zero abstraction overhead for Kernel execution!

Heterogenous Programming: Using different backends in a synchronized manner



(Slide32) Lastly 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. I am also willing to do a hands on session on alpaka programming if there is an interest, please let us now. Now we are going to move on to discussion session. Before moving on; I would like to ask to alpaka team if they want to add anything to what I presented.