CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

Alpaka Parallel Programming Library
In a Nutshell

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

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 in a Nutshell

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

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 H100 GPU

4 Fugaku (Japan) 442 Petaflop/s, Fujitsu A64FX CPU

5 Lumi (Finland) 380 Petaflop/s, AMD EPYC CPU + AMD Instinct GPU

6 Leonardo (Italy) 238.7 Petaflop/s, Intel Xeon CPU + NVIDIA A100 GPU

7 MareNostrum (Spain) 183.2 Petaflops/s, Intel Xeon CPU + NVIDIA H100 GPU

8 Summit(USA) 148.3 Petaflops/s, IBM Power9 CPU + NVIDIA Volta GPU

9 Eos(USA) 121.4 Petaflops/s, Intel Xeon CPU + NVIDIA H100 GPU

10 Sierra(USA) 94 Petaflops/s, IBM POWER9 CPU + NVIDIA Tesla V100 GPU

www.top500.org

THE LIST

11/2023 Highlights

The 62nd edition of the TOP500 shows five new or upgraded entries in the top 10 but the Frontier system still remains the only true exascale machine with an HPL score of 1.194 Exaflop/s.

The Frontier system at the Oak Ridge National Laboratory, Tennessee, USA remains the No. 1 system on the TOP500 and is still the only system reported with an HPL performance exceeding one Exaflop/s. Frontier brought the pole position back to the USA one year ago on the June 2022 listing and has since been remeasured with an HPL score of 1.194 Exaflop/s.

Frontier is based on the latest HPE Cray EX235a architecture and is equipped with AMD EPYC 64C 2GHz processors. The system has 8,699,904 total cores, a power efficiency rating of 52.59 gigaflops/watt, and relies on HPE's Slingshot 11 network for data transfer.

The Aurora system at the Argonne Leadership Computing Facility, Illinois, USA is currently being commissioned and will at full scale exceed Frontier with a peak performance of 2 Exaflop/s. It was submitted with a measurement on half of the final system achieving 585 Petaflop/s on the HPL benchmark which secured the No. 2 spot on the TOP500.

Aurora is built by Intel based on the HPE Cray EX - Intel Exascale Compute Blade which uses Intel Xeon CPU Max Series processors and Intel Data Center GPU Max Series accelerators which communicate through HPE's Slingshot-11 network interconnect.

The Eagle system installed in the Microsoft Azure cloud in the USA is newly listed as No. 3. This Microsoft NDv5 system is based on Intel Xeon Platinum 8480C processors and NVIDIA H100 accelerators and achieved an HPL score of 561 Pflop/s.

read more »

List Statistics

Vendors System Share

1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE
6	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN
7	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-

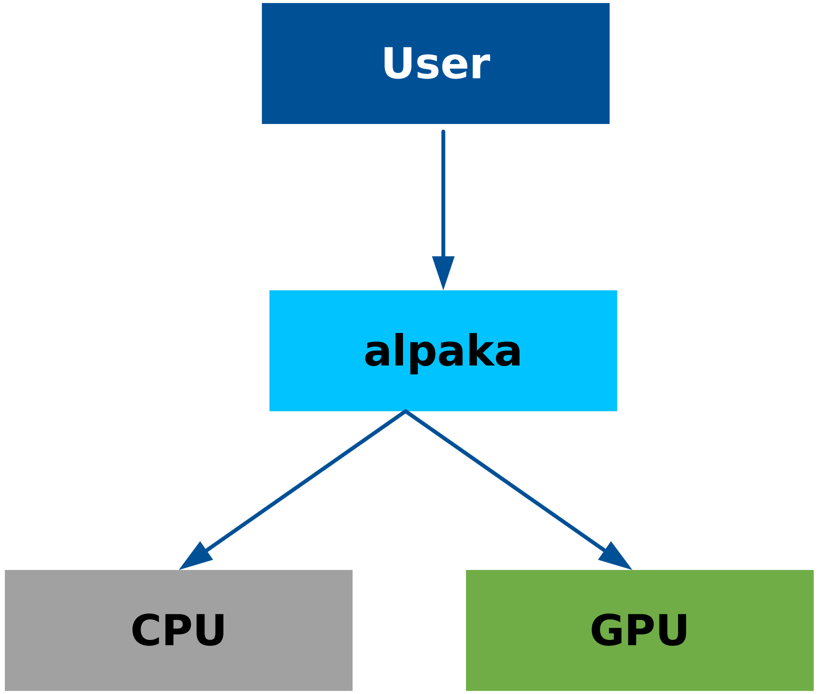
alpaka In a nutshell

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 platforms
  - No default device, default queue, built-in functions, language extentions
- **Easy change of the backend**
  - Code needs only minor adjustments to support different accelerators
- **Direct usage of vendor APIs, not depend on "unified APIs"**
  - GPU Backends: Hip backend uses HIP API, Cuda backend uses Cuda API, SYCL backend can be used for Intel GPUs
  - Cuda profilers and HIP profilers can be used to profile Alaka generated kernel code.
  - CPU Backends: OpenMp, Threads, TbbBlocks
- **Easy indexing of threads in kernels**
- **Alpaka proposes and validates the type of parallelism**
- **Heterogenous Programming**: Using different backends in a synchronized manner



alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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/**

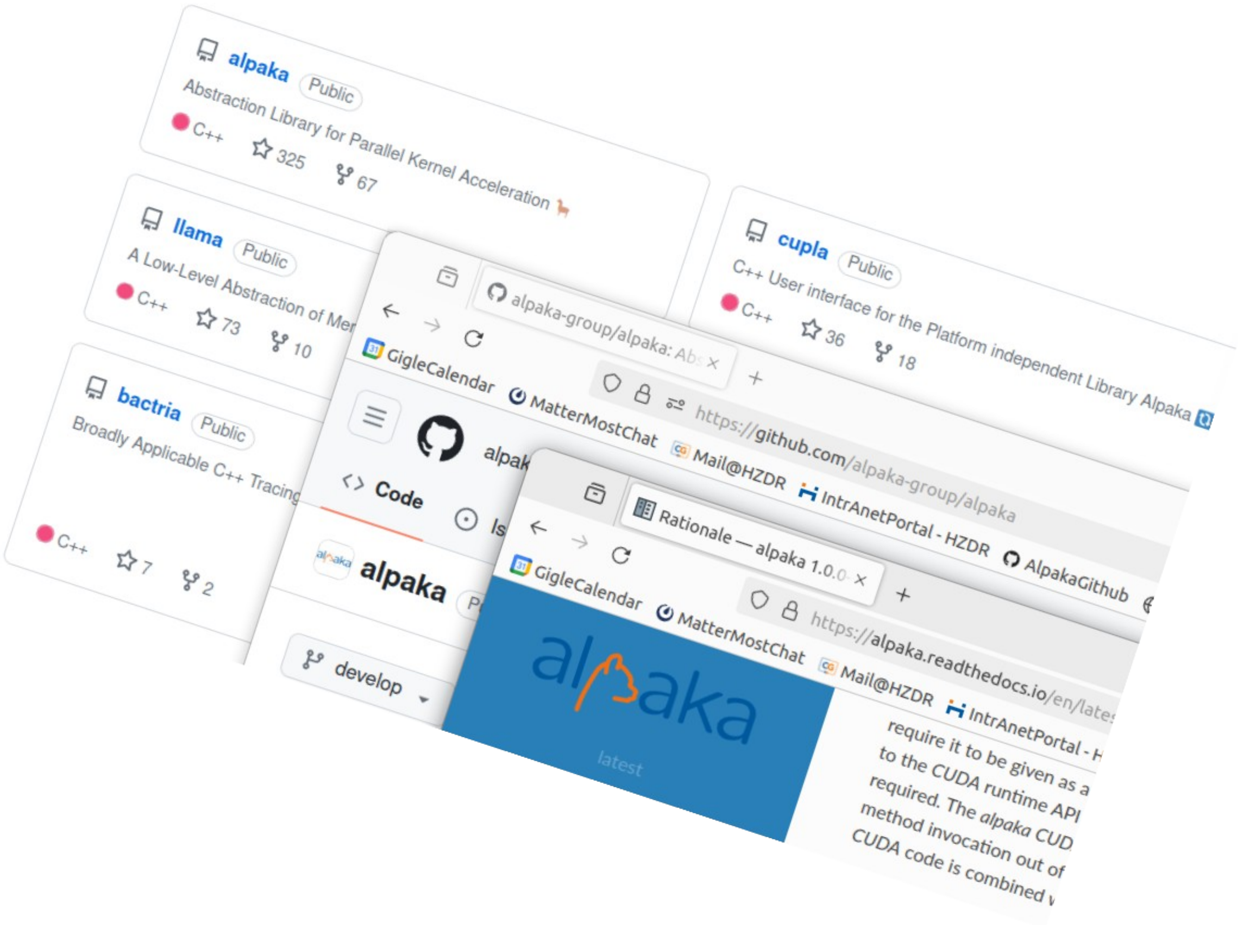
- Installation guide
- Cheatsheet
- Abstraction model and the rationale behind alpaka

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



alpaka in a Nutshell

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, MS VS)
- Portable across operating systems: Linux, macOS, Windows are supported

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Installing alpaka [shorten]

- alpaka requires Boost, Cmake and a modern C++ compiler (`g++`, `clang++`, Visual C++, …)
  - Linux:
    - `sudo dnf install boost-devel` (RPM)
    - `sudo apt install libboost-all-dev` (DEB)
  - macOS:
    - `brew install boost` (Using Homebrew, **https://brew.sh**)
    - `sudo port install boost` (Using MacPorts, **https://macports.org**)
  - Windows: `vcpkg install boost` (Using vcpkg, **https://github.com/microsoft/vcpkg**)
- Depending on your target platform you may need additional packages
  - NVIDIA GPUs: CUDA Toolkit (**https://developer.nvidia.com/cuda-toolkit**)
  - AMD GPUs: ROCm / HIP (**https://rocmdocs.amd.com/en/latest/index.html**)
- CMake is the preferred system for building and installing
  - Linux: `sudo dnf install cmake` (RPM) or `sudo apt install cmake` (DEB)
  - macOS and Windows: Download the installer from https://cmake.org/download/
- **Download alpaka: `git clone -b develop https://github.com/alpaka-group/alpaka.git`**

Install alpaka's dependencies:

- 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`
- In the build directory use CMake for configuration (replace the install prefix with an actual path):
  `cmake -DCMAKE_INSTALL_PREFIX=/some/other/path/ ..`
  - Alpaka determines the installed backends and makes the configuration of Cmake variables accordingl
  - Ccmake tool would show the cmake variables configured by the command above. You can change cmake c
  `ccmake ../`
- Your alpaka installation will later reside in /some/other/path/
- If you are content with the default directories you can omit the install prefix
  - This requires administrator privileges!
  - This will result in a system-wide installation!
  - On Linux and macOS the default directory will be /usr/local
  - On Windows the default directory will be C:\Program Files
- Execute the installation command (note the dot):
  `cmake --install .`
- You should now have a complete alpaka installation in the directory you chose earlier
- Did you encounter problems or errors?
  - Check the documentation: **https://alpaka.readthedocs.io/en/latest/install/instructions.html**
  - Still having problems? Please report them here: **https://github.com/alpaka-group/alpaka/issues**

```
BUILD_TESTING                    ON
Boost_ATOMIC_LIBRARY_RELEASE     /usr/lib/x86_64-linux-gnu/libboost_atomic.so.
Boost_INCLUDE_DIR                /usr/include
CMAKE_BUILD_TYPE                 RELEASE
CMAKE_CUDA_ARCHITECTURES         52
CMAKE_INSTALL_PREFIX             /install
FETCHCONTENT_BASE_DIR            /home/yusufo81/projects/alpaka-dir/alpaka/bui
FETCHCONTENT_FULLY_DISCONNECTE   OFF
FETCHCONTENT_QUIET               ON
FETCHCONTENT_SOURCE_DIR_MDSPAN
FETCHCONTENT_UPDATES_DISCONNEC   OFF
FETCHCONTENT_UPDATES_DISCONNEC   OFF
MDSPAN_CXX_STANDARD              DETECT
MDSPAN_ENABLE_BENCHMARKS         OFF
MDSPAN_ENABLE_COMP_BENCH         OFF
MDSPAN_ENABLE_CONCEPTS           ON
MDSPAN_ENABLE_CUDA               OFF
MDSPAN_ENABLE_EXAMPLES           OFF
MDSPAN_ENABLE_OPENMP             ON
MDSPAN_ENABLE_TESTS              OFF
MDSPAN_USE_SYSTEM_GTEST          OFF
RT_LIBRARY                       /usr/lib/x86_64-linux-gnu/librt.a
TBB_DIR                          /usr/lib/x86_64-linux-gnu/cmake/TBB
alpaka_ACC_CPU_B_OMP2_T_SEQ_EN   OFF
alpaka_ACC_CPU_B_SEQ_T_OMP2_EN   OFF
alpaka_ACC_CPU_B_SEQ_T_SEQ_ENA   ON
alpaka_ACC_CPU_B_SEQ_T_THREADS   OFF
alpaka_ACC_CPU_B_TBB_T_SEQ_ENA   OFF
alpaka_ACC_CPU_DISABLE_ATOMIC_   OFF
alpaka_ACC_GPU_CUDA_ENABLE       ON
alpaka_ACC_GPU_CUDA_ONLY_MODE    OFF
alpaka_ACC_GPU_HIP_ENABLE        OFF
alpaka_ACC_GPU_HIP_ONLY_MODE     OFF
alpaka_ACC_SYCL_ENABLE           OFF
alpaka_ASSERT_ACC_ENABLE         ON
alpaka_BLOCK_SHARED_DYN_MEMBER   47
alpaka_BUILD_BENCHMARKS          ON
alpaka_BUILD_EXAMPLES            ON
```

alpaka in a Nutshell

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Accessing alpaka examples [shorten]

- There are many examples in the `example` directory! You can build them all at once from your `build` directory:
  ```
  cmake -Dalpaka_BUILD_EXAMPLES=ON -DCMAKE_BUILD_TYPE=Release ..
  cmake --build . --config Release
  ```
- `Switch to the vectorAdd example (from the top-level source directory):`
  ```
  cd example/vectorAdd
  mkdir build
  cd build
  ```
- `If you installed alpaka to a standard location (= default CMAKE_INSTALL_PREFIX):`
  ```
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```
- `If you installed alpaka to a non-standard location:`
  ```
  cmake -DCMAKE_BUILD_TYPE=Release -Dalpaka_ROOT=/path/to/installed/alpaka ..
  ```
- Alternative: Set the `CMAKE_PREFIX_PATH` environment variable (note the missing `alpaka`):
  - Linux / macOS: `export CMAKE_PREFIX_PATH=/path/to/installed`
  - Windows: `set CMAKE_PREFIX_PATH=C:\path\to\installed`

- `Switch to the vectorAdd example (from the top-level source directory):`
  ```
  cd example/vectorAdd

  mkdir build

  cd build
  ```
- `If you installed alpaka to a standard location (= default CMAKE_INSTALL_PREFIX):`
  ```
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```
- `If you installed alpaka to a non-standard location:`
  ```
  cmake -DCMAKE_BUILD_TYPE=Release -Dalpaka_ROOT=/path/to/installed/alpaka ..
  ```
- Alternative: Set the `CMAKE_PREFIX_PATH` environment variable (note the missing `alpaka`):
  - Linux / macOS: `export CMAKE_PREFIX_PATH=/path/to/installed`
  - Windows: `set CMAKE_PREFIX_PATH=C:\path\to\installed`

- `Build the example (note the dot):`
  ```
  cmake --build . --config Release
  ```
- `If everything worked correctly you should see the executable somewhere in your build tree:`
  ```
  ls
  ```
  - `Linux / macOS: CMakeCache.txt CMakeFiles cmake_install.cmake Makefile` **`vectorAdd`**
  - Windows: The executable should be in the `Release` subdirectory of `build`. `cd` to the `Release` directory.
- `Execute the example:`
  ```
  ./vectorAdd
  ```
  - Expected output: `Execution results correct!`

Tenets of Thread-Parallel Programming

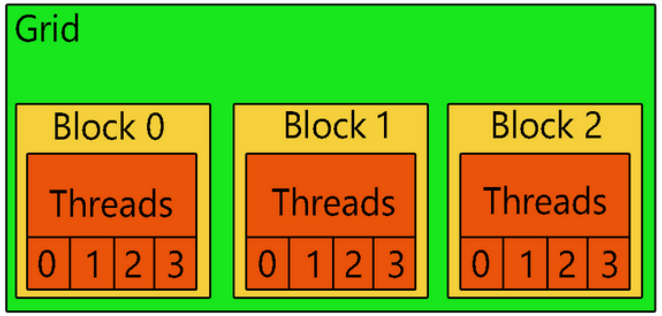
- **Massive number of threads.** Large number of threads should run the same code seperately! This code is called “Kernel”.
- **Indexing of threads.** Each thread should work on a different data portion or do a specific task. Hence, each thread has an index, inside the kernel this index is accessable.
- **Grouping threads to map them to the cores.** Threads are grouped in blocks *to map them to the cores to increase occupancy*.

  The warpsize of the GPUs and memory(shared memory,registers) usage of kernel affects the performance. Therefore grid of threads are devided into thread blocks.

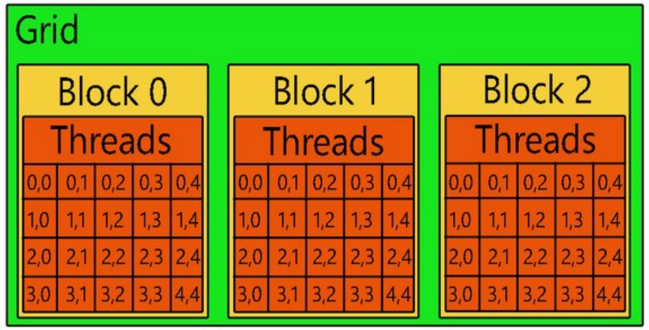
  Extent: A vector representing the sizes along each dimension. In 3d, a vector of 3 items {height,width,depth}

  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{1,3}
Block Thread Extent: Vec{5,5}

Vector Addition Problem

-Fix the number of data dimensions and the index type

-Create A and B vectors on Host
Use dynamic array **alpaka::Buf**

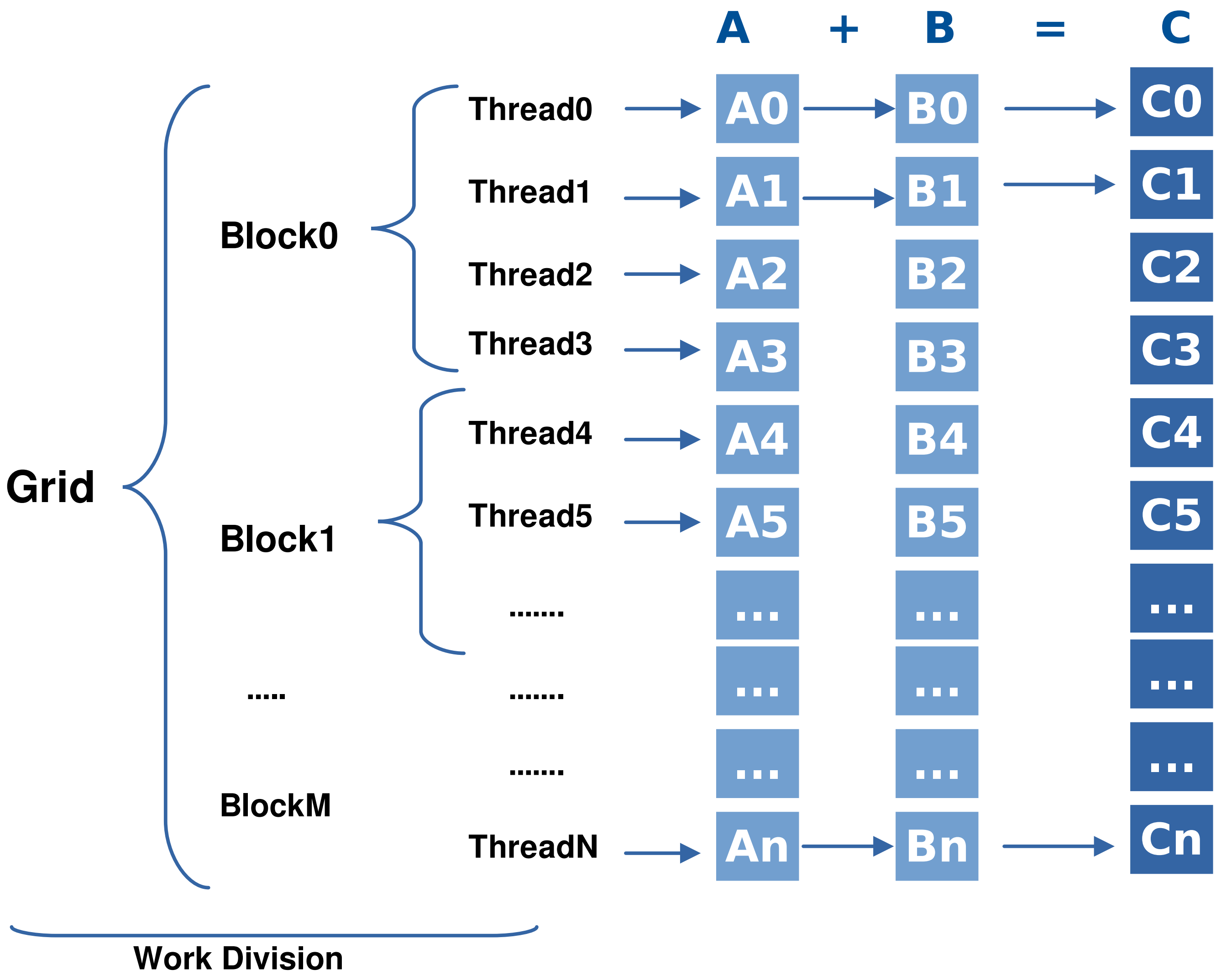
-Create empty C vector on Host
Use alpaka::Buf

-Copy A and B Vectors to Device
Use **alpaka::memcpy**

-Determine the Paralelism: Work Division. Extents of threads per block, blocks per grids, elements per thread.

-Create the Queue
Use **alpaka::Queue**

-Execute the kernel on device with the workdiv on the queue. Pass the A,B,C vectors to the executer. Use **alpaka::exec**



Parallel vector addition code

- Define kernel
- Select the backend depending on your system, the accelerator!
- Select the device (e.g your specific GPU)
- Decide on proper WorkDivision
- Create input vectors A and B and result vector C on host (e.g CPU)
- Create input vectors A and B and result vector C on device (e.g CPU)
- Copy the vectors to the device memory
- Run kernel
- Copy the result vector back to the host vector

```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);

    // 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!
    alpaka::wait(queue);
}
```

Define the Alpaka Kernel

- Contains the algorithm
- Written on per-data-element basis
- It allows access to the thread index directly
- alpaka Kernels are functors (function-like C++ structs / classes)
- `operator()` is annotated with `ALPAKA_FN_ACC` specifier
- `operator()` must return `void`
- `operator()` must be `const`
- `Arguments can be pointers and trivially copyable types`
- `Agnostic to device details`
- `Code piece that is run by each thread`

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

Obtaining the indices and extents in Kernel

- **alpaka provides API functions for obtaining indices**
  - Index of Thread on the Grid: `alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[dim];`
  - Index of Thread on a Block: `alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc)[dim];`
  - Index of Block on the Grid: `alpaka::getIdx<alpaka::Grid, alpaka::Blocks>(acc)[dim];`

- **alpaka provides API functions for obtaining extents of the grid and the blocks**
  - Number of Threads on the Grid: `alpaka::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc)[dim];`
  - Number of Threads on a Block: `alpaka::getWorkDiv<alpaka::Block, alpaka:Threads>(acc)[dim];`
  - Number of Blocks on the Grid: `alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(acc)[dim];`

alpaka in a Nutshell: Accelerator

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Select the Accelerator

- alpaka provides a number of pre-defined Accelerators.
- Accelerator is a type that is only instantiated on Device not on the host.
- Accelerator is an abstraction that makes the kernel ‘device agnostic’. (Gives kernel means the access thread index. Responsible for the thread state, memory management, and synchronization ??)
- For GPUs:
  - `AccGpuCudaRt` for NVIDIA GPUs
  - `AccGpuHipRt` for AMD, Intel and NVIDIA GPUs
- For CPUs
  - `AccCpuFibers` based on Boost.fiber
  - `AccCpuOmp2Blocks` based on OpenMP 2.x
  - `AccCpuOmp4` based on OpenMP 4.x
  - `AccCpuTbbBlocks` based on TBB
  - `AccCpuThreads` based on `std::thread`

```
// Easy change of the backend type!
// Example: CUDA GPU accelerator
using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;

// Switch to AMD: HIP GPU accelerator
using Acc = alpaka::AccGpuHipRt<Dim, Idx>;

// Switch to CPU Omp: CPU accelerator
using Acc = alpaka::AccCpuOmp2Blocks<Dim, Idx>;
```

```
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);

    // 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
    using DevHost = alpaka::DevCpu;
    // Allocate 3 host memory buffers
    using BufHost = alpaka::Buf<DevHost, DataType, Dim, Idx>;
    ...
    ...
        // Instantiate the kernel function object
    VectorAddKernel kernel;
    alpaka::exec<Acc>( // Run the kernel execution task
        queue,
```

alpaka in a Nutshell

Get device instance

alpaka Devices

- Each alpaka Device instance represents a single physical device
- Determined by programmer's Accelerator choice
- Easy management of physical devices
- Contains device information

```
using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
using DevAcc = alpaka::Dev<Acc>;

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

auto const name = alpaka::getName(devAcc);          // Back-end-defined device name
auto const bytes = alpaka::getMemBytes(devAcc);     // Size of device memory
auto const free = alpaka::getFreeMemBytes(devAcc); // Size of available device memory
// Provides the means for device management:
alpaka::reset(devAcc);
```

```
// Select a device
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!
    elementsPerThread, false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

// Get the host device for allocating memory on the host.
using DevHost = alpaka::DevCpu;
auto const platformHost = alpaka::PlatformCpu{};
auto const devHost = alpaka::getDevByIdx(platformHost, 0);

// Allocate 3 host memory buffers
using BufHost = alpaka::Buf<DevHost, Data, Dim, Idx>;
BufHost bufHostA(alpaka::allocBuf<Data, Idx>(devHost, extent));
BufHost bufHostB(alpaka::allocBuf<Data, Idx>(devHost, extent));
BufHost bufHostC(alpaka::allocBuf<Data, Idx>(devHost, extent));

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, 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));

// Define the synchronization behavior of a queue
// choose between Blocking and NonBlocking
using QueueProperty = alpaka::Blocking;
// Create a queue on the device
alpaka::Queue<Acc, QueueProperty> queue(devAcc);
```

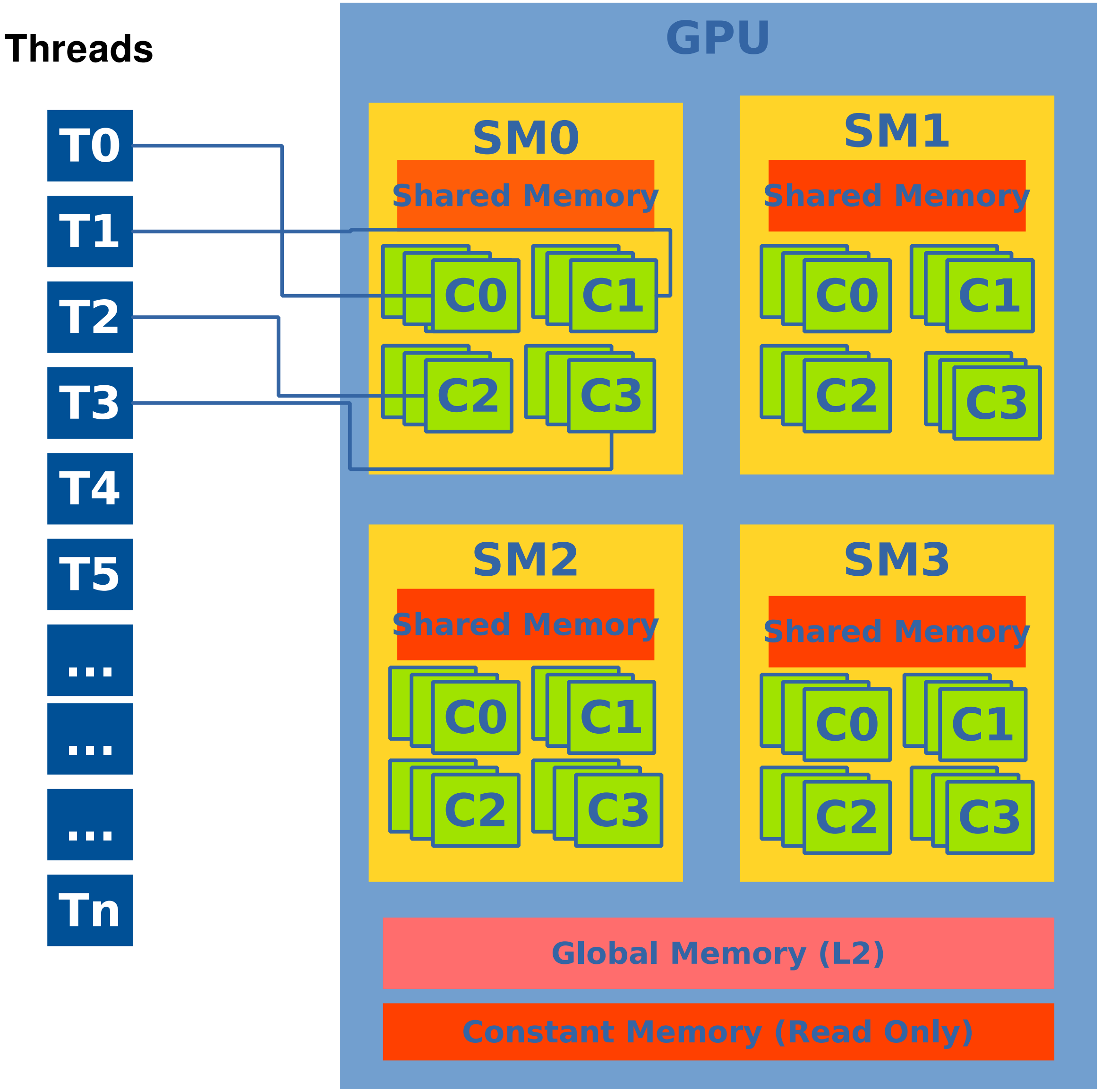
alpaka

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Determine the WorkDivision-I

How to distribute threads among SMs?

- Main determinants of mapping threads to the SMs:
  - Number of cores SM,
  - Warpsize,
  - Register and local memory (lm) usage of each thread,
  - Shared memory usage of each thread,
  - Threads per SM, Threads per Block, Blocks per Device
- Memory latencies: Global Memory and Constant memory has different latencies.
- Memory sizes: Size of shared memory used by threads in a block or blocks assigned to an SM



alpaka in a Nutshell

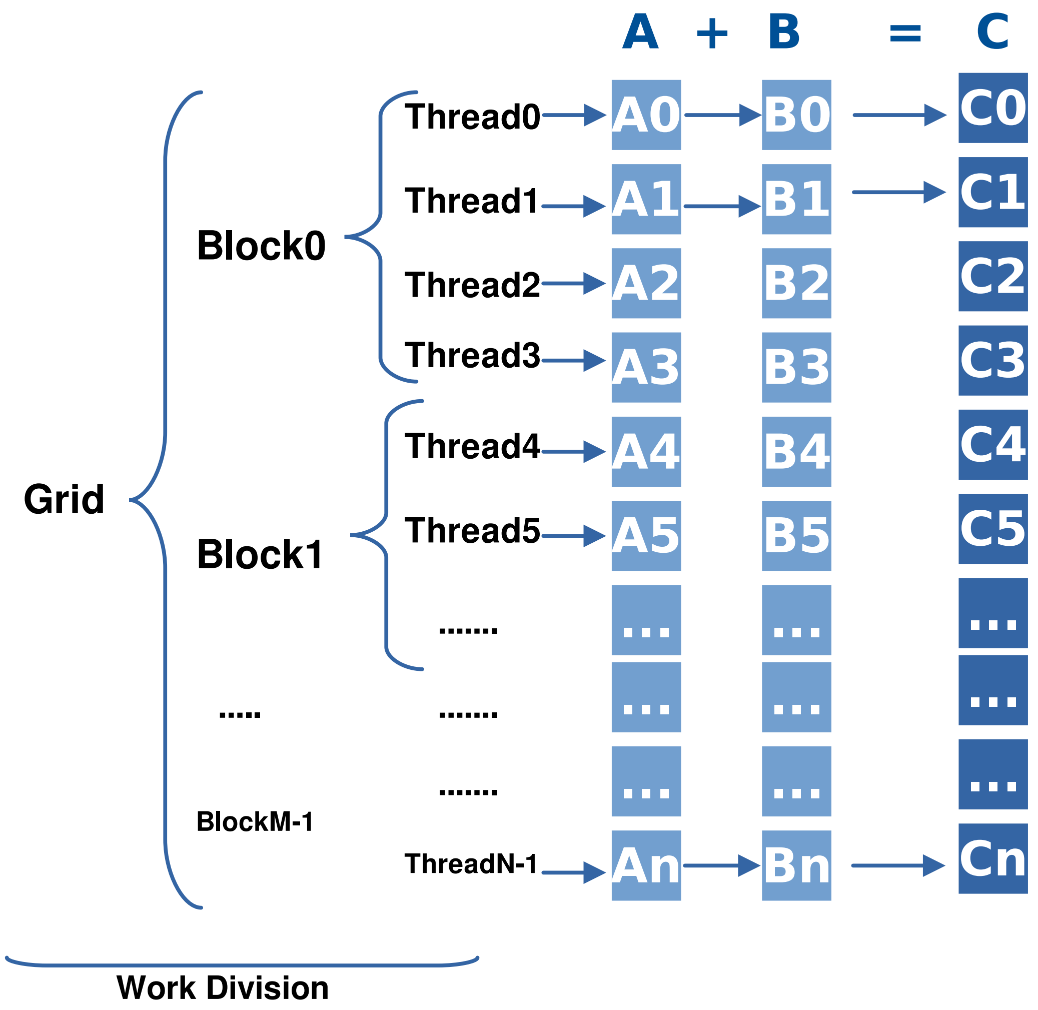
Determine the workdivision-II

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

    `Vec{M}` or `Vec{1,1,M}` depending on the number of dimensions.
  - Block thread extent.

    `Vec{4}` or `Vec{1,1,4}`.
  - Elements per thread
- Setting work-div manually

```
using Dim1D = alpaka::DimInt<1>;//Set number of dims to 1
using Vec1D = alpaka::Vec<Dim1D, Idx>;//Define alias
auto workDiv1D = alpaka::WorkDivMembers(Vec1D{M},  Vec1D{4u},  Vec1D{1u});
// alternatively
using Dim3D = alpaka::DimInt<3>;//Set number of dims to 3
using Vec3D = alpaka::Vec<Dim3D, Idx>; //Define alias
auto workDiv3D = alpaka::WorkDivMembers(Vec3D{1,1,M},  Vec3D{1,1,4u},  Vec3D{1,1,1u});
```



alpaka

1D WorkDivision!

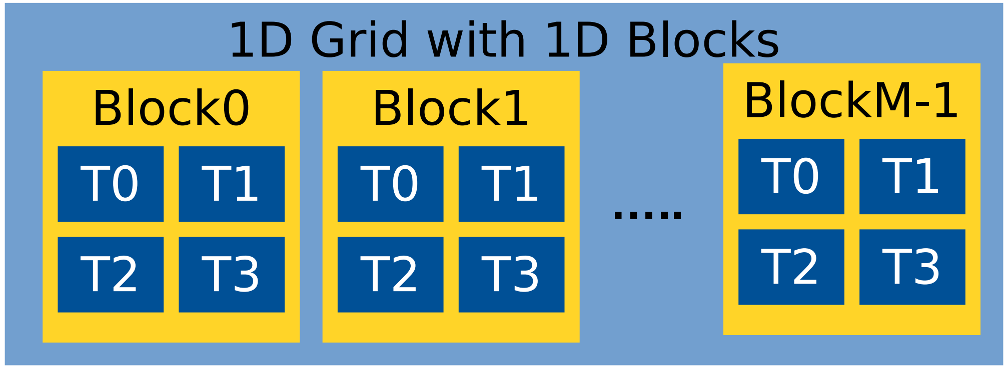
Grid Block Extent 1D vector = Vec{M}
Block Thread Extent 1D vector = Vec{4u}
Thread Elem Extent 1D vector = Vec{1u}
Grid Thread Extent 1D vector =Vec{4*M} or Vec{N}

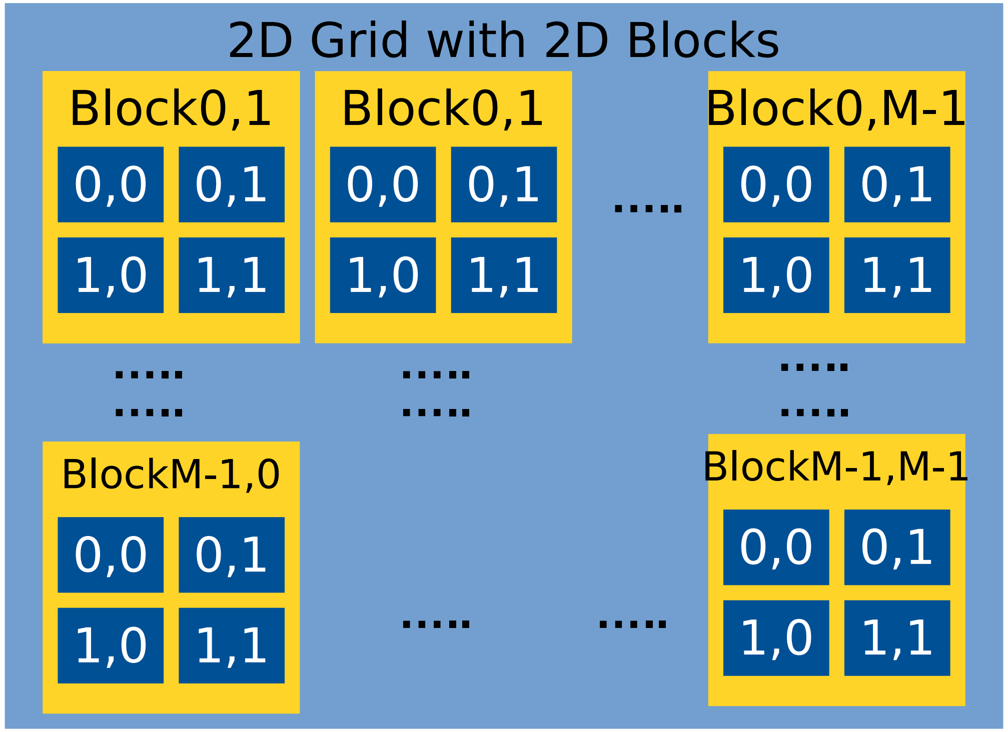
1D WorkDivision {{M},{4u},{1u}}
// if Dim is 3 then fill with 1u
1D WorkDivision {{1u,1u,M},{1u,1u,4},{1u,1u,1u}}

2D WorkDivision!

Grid Block Extent 2D vector = Vec{M,M}
Block Thread Extent 2D vector = Vec{2u,2u}
Thread Elem Extent 2D vector = Vec{1u,1u}
Grid Thread Extent 2D vector =Vec{4u*M,4u*M} or Vec{N,N}

2D WorkDivision {{M,M},{2u,2u},{1u,1u}}
2D WorkDivision {{1u,M,M},{1u,2u,2u},{1u,1u,1u}}





alpaka in a Nutshell

Determine the WorkDivision-III

- Use **`getValidWorkDiv`** function
- Inputs are
  - Full grid-thread extent. Hence we guarantee total number of threads needed.
  - Elements per thread extent
- **`getValidWorkDiv`** generates devides the full grid-thread extent to blocks of given dimension.

```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

Allocate data vectors on host and device. Copy data vectors to the Device

- **`alpaka::Buf`** is multi dimensional dynamic array for all devices, copyable among host and accelerator devices.
- **`alpaka::allocBuf`** allocates memory to the given device.
- **`alpaka::memcpy`** copies the data from one buffer to another. User should not have to specify which buffer resides on which device.
- `alpaka::Buf` is device-type bound. (Host device or Acc device)
- Memory copy needs to be synchronized , therefore needs a queue

```cpp
// 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!
    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);
```

alpaka in a Nutshell

Create the Queue for memcpy and kernel executions

- Alpaka::Queue provides communication between the host and the device
- Queue of tasks on the device
- Queue is always FIFO
- Create **alpaka::Queue** using the device instance and the accelerator type (e.g GPU)
- Two queue types: blocking and non-blocking
- Blocking queues block the Host until Device-side command returns.
- Non-blocking queues return control to Host immediately, Device-side command runs asynchronously

```
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!
alpaka::wait(queue);
// bufHostC is filled with the result!
```

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Execute the kernel using the workdiv, copy result back to the Host

- **alpaka::exec** function is used
- Result is stored in an **alpaka::Buf**
- Copy the result in device to the host
- If the queue not blocking then we can use **alpaka::wait** on the queue for sychronization.

```
// 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!
alpaka::wait(queue);
```

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

On Tasks and Events

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

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

- Order of tasks in different queues is unspecified
  - ```
    alpaka::enqueue(queueA, task1);
    alpaka::enqueue(queueB, task2); // task2 starts before, after or in parallel to task1
    ```
- For easier synchronization, alpaka Events can be inserted before, after or between Tasks:

```
auto myEvent = alpaka::Event<alpaka::Queue>(myDev);

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

Programming Heterogeneous Systems-I

Using multiple Platforms Synchronously

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

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

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

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

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

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Programming Heterogeneous Systems-II

Communication by Buffers

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

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

/* Initialization … */

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

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

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

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

Tips

- **If you want to pass pointers of views generated from data with numdim 2d or larger, you need to consider pitch**
- If there are unused number of dimentions in workdiv use 1 for that dimension
- CPU to GPU copies require GPU Queue
- 2 queues synchorinization

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Summary of Alpaka Structures

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

[ToBeChanged] CUPLA: Converting CUDA code to Alpaka

- Change the suffix *.cu of the CUDA source files to ***.cpp***
- Remove cuda specific includes on top of your header and source files
- Add ***#include <cuda_to_cupla.hpp>*** remove *#include <cuda_runtime.h>*
- Tansform kernels (__global__ functions) to functors
- The functor's ***operator()*** must be qualified as ***const***
- Add the function prefix ***ALPAKA_FN_ACC*** to the operator() const
- Add as first (templated) kernel parameter the accelerator with the name acc (it is important that the accelerator is named acc because all cupla-to-alpaka replacements use the variable acc)
- If the kernel calls other functions you must pass the accelerator ***acc*** to each call
- Add the qualifier ***const*** to each parameter which is not changed inside the kernel

- **CUDA kernel**

```
template< int blockSize >
__global__ void fooKernel( int * ptr, float value )
{
    // ...
}
```

Cupla kernel

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

CUDA kernel call at host

```
dim3 gridSize(42,1,1);
dim3 blockSize(256,1,1);
// extern shared memory and stream is optional
fooKernel< 16 ><<< gridSize, blockSize, 0, 0 >>>( ptr, 23 );
```

CUPLA kernel call at host:

```
dim3 gridSize(42,1,1);
dim3 blockSize(256,1,1);
// extern shared memory and stream is optional
CUPLA_KERNEL(fooKernel< 16 >)( gridSize, blockSize, 0, 0 )( ptr, 23 );
```

CUDA shared memory (in kernel)

```
__shared__ int foo;
__shared__ int fooCArray[32];
__shared__ int fooCArray2D[4][32];
// extern shared memory (size was defined during the host side kernel call)
extern __shared__ float fooPtr[];

int bar = fooCArray2D[ threadIdx.x ][ 0 ];
```

CUPLA shared memory (in kernel)

```
sharedMem( foo, int );
sharedMem( fooCArray, cupla::Array< int, 32 > );
sharedMem( fooCArray, cupla::Array< cupla::Array< int, 4 >, 32 > );
sharedMemExtern( fooPtr, float );

int bar = fooCArray2D[ threadIdx.x ][ 0 ]
```

Use ALPAKA_FN_ACC in device function definitions and add an acc parameter.
(Note that to be exact the acc parameter is only necessary when alpaka functions like blockIdx or atomicMax etc are used.)

- **CUDA Device Function**

```
template< typename T_Elem >
__device__ int deviceFunction( T_Elem x )
{
    // ...
}
```

CUPLA

```
template< typename T_Acc, typename T_Elem >
ALPAKA_FN_ACC int deviceFunction( T_Acc const & acc, T_Elem x )
{
    // ...
}
```

- **CUDA Device Function Call**

```
auto result = deviceFunction( x );
```

CUPLA

```
auto result = deviceFunction( acc, x );
```

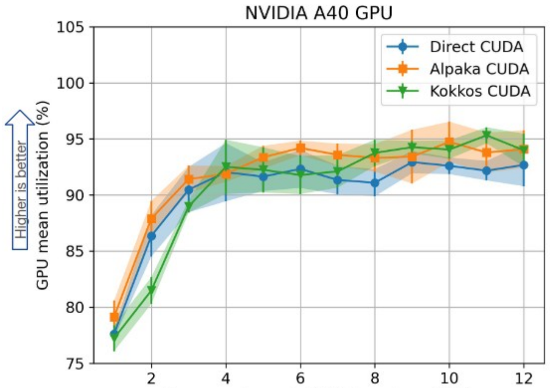
How to start using Alpaka

- Use Alpaka directly using examples and the cheat-sheet
- Don’t write code initially on Cuda because alpaka is already low level, like Cuda.
- If you have a codebase in cuda, converting to Cupla can be a fast solution to benefit from alpaka features!

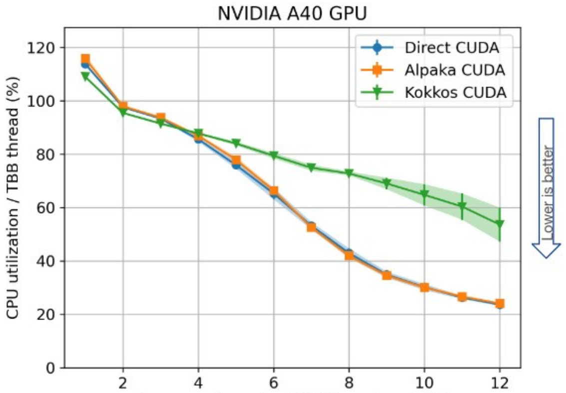
alpaka in a Nutshell

Alpaka Performance

Mean GPU and CPU utilization on NVIDIA A40 GPU

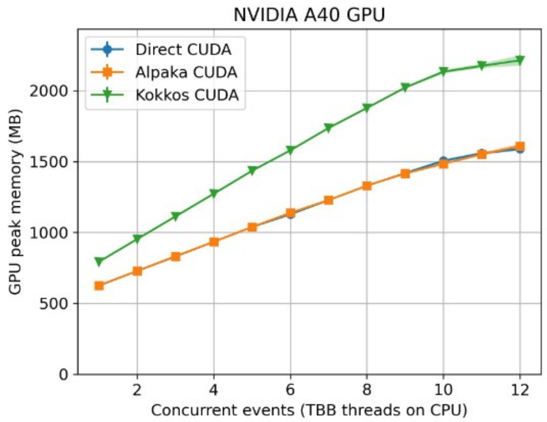


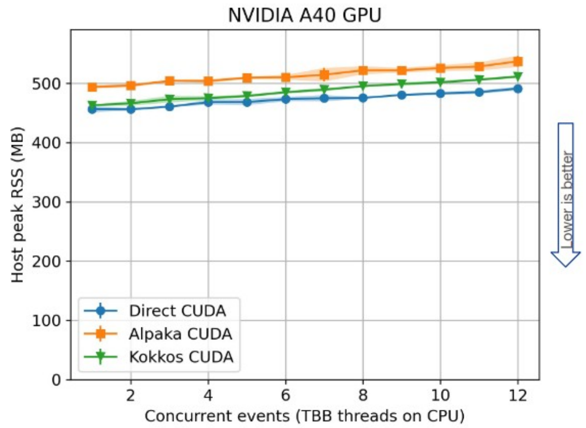
As reported by nvidia-smi, mean from samples



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.

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

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Community and Long Term Support

- Partners using and contributing to Alpaka

CERN

HZB Helmholtz Zentrum Berlin

DLR Deutsches Zentrum für Luft- und Raumfahrt

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

- Alpaka is a part of Helmholtz Roadmap 2027-2034

Alpaka in the wild…

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

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

PIConGPU

alpaka In a nutshell

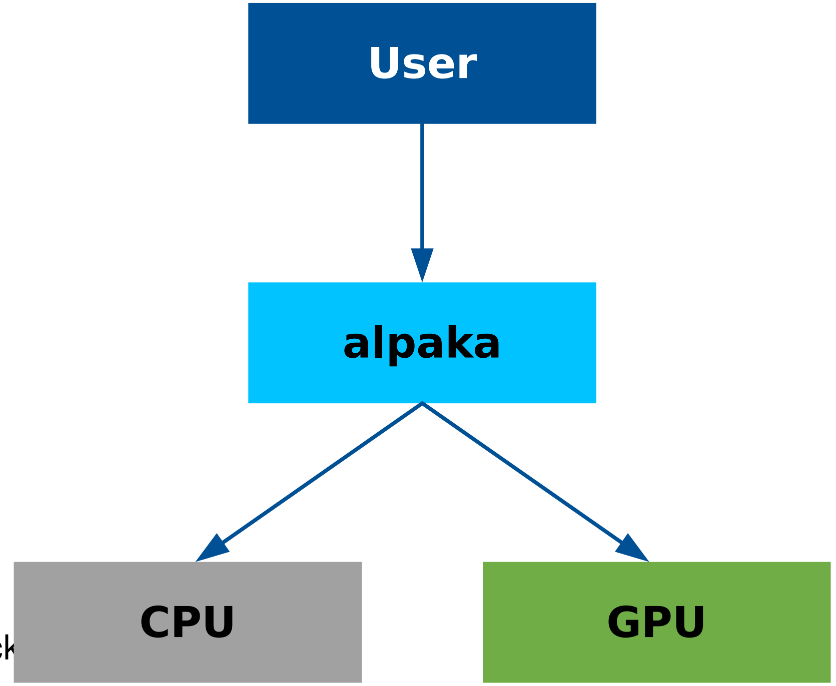
As a summary

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**
  - Code needs only minor adjustments to support different accelerators
- **Easy indexing of threads in kernels**
- **Easy setup of the type of parallelism by WorkDivision** (Block sizes in grid, Thread sizes in block
- **Heterogenous Programming**: Using different backends in a synchronized manner



CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

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

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

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

Appendix

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

[Appendix1] WorkDivision Example

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

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

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

// Instantiate the kernel function object
HelloWorldKernel helloWorldKernel;

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

[Appendix2] Matrix Vector Multiplication Example

-

alpaka Accelerator Details

APPENDIX-5

- Accelerator chosen by the programmer and **hides hardware specifics** behind alpaka's abstract API
  ```
  using Acc = acc::AccGpuCudaRt<Dim, Idx>;
  ```
- **Inside Kernel:** contains thread state, provides access to alpaka's device-side API
  - **The Accelerator provides the means to access to the indices**
    ```
    // get thread index on the grid
    auto gridThreadIdx = alpaka::getIdx<Grid, Threads>(acc);
    // get block index on the grid
    auto gridBlockIdx = alpaka::getIdx<Grid, Blocks>(acc);
    ```
  - **The Accelerator gives access to alpaka's shared memory** (for threads inside the same block)
    ```
    // allocate a variable in block shared static memory
    auto & mySharedVar = block::shared::st::allocVar<int, __COUNTER__>(acc);
    ```
    ```
    // get pointer to the block shared dynamic memory
    float * mySharedBuffer = block::shared::dyn::getMem<float>(acc);
    ```
  - **It also enables synchronization on the block level**
    ```
    // synchronize all threads within the block
    block::sync::syncBlockThreads(acc);
    ```
  - **Internally, the accelerator maps all device-side functions to their native counterparts**
    - Device-side functions require the accelerator as first argument:

      `math::sqrt(acc, /* … */);` `time::clock(acc);`

      `atomic::atomicOp<atomic::op::Or>(acc, /* … */, hierarchy::Grids);` (Atomics)
- **On Host:** Meta-parameter for choosing correct physical device and dependent types

APPENDIX-6 Physical device information and management by “alpaka Device”

- Each alpaka Device represents a single physical device;
- Contains device information:
  - `auto const name = alpaka::getName(myDev);         // Back-end-defined device name`
  - `auto const bytes = alpaka::getMemBytes(myDev);    // Size of device memory`
  - `auto const free = alpaka::getFreeMemBytes(myDev); // Size of available device memory`
- Provides the means for device management:
  - `alpaka::reset(myDev);                             // Reset GPU device state`
- Encapsulates back-end device:
  - `auto nativeDevice = alpaka::getDev(myDev);        // nativeDevice is not portable!`

APPENDIX-7 Queue operations

- Queues execute Tasks (see next slide):
  - `alpaka::enqueue(myQueue, taskRunKernel);`
- Check for completion:
  - `bool done = alpaka::empty(myQueue);`
- Wait for completion, Events (see next slide), or other Queues:
  - `alpaka::wait(myQueue);              // blocks caller until all operations have completed`
  - `alpaka::wait(myQueue, myEvent);     // blocks myQueue until myEvent has been reached`
  - `alpaka::wait(myQueue, otherQueue); // blocks myQueue until otherQueue’s ops have completed`

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Appendix8 - Changing the target platform

```
using namespace alpaka;

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

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

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

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

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

APPENDIx 9

Acc versus Device

Acc is a host side compile time construct but instantiated in Kernel Abstraction that make the kernel device agnostic