CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

Alpaka Parallel Programming Library In a Nutshell

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

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

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

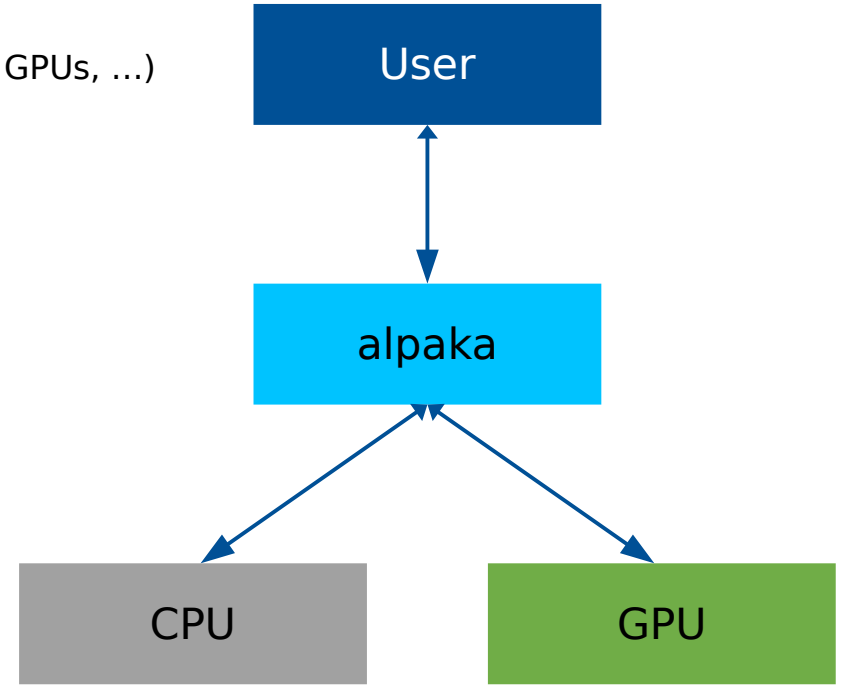
10 Sierra(USA) 94 Petaflops/s, IBM POWER9 CPU + NVIDIA Tesla V100 GPU

alpaka In a nutshell

alpaka's purpose

Without alpaka

- Hardware ecosystem is heterogenous and multiple hardware types commonly used (CPUs, GPUs, …)
- Platforms 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 variables or functions. No language extentions
- **Easy change of the backend**
  - Code needs only minor adjustments to support different accelerators
- **Direct usage of vendor APIs**
- **Easy indexing of threads in kernels**
- **Proposition and validation of type of parallelism** (Block sizes in grid, Thread sizes in block…)
- **Heterogenous Programming**: Using different backends in a synchronized manner

alpaka in a Nutshell: Accelerator

Switching the Accelerator Easily

- alpaka provides a number of pre-defined Accelerators in the `acc` namespace.
- 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`

```
// 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>;
```

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Programming with alpaka

- C++ only!
- Header-only library: No additional runtime dependency introduced
- Modern library: alpaka is written entirely in C++17
- Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MS Visual Studio)
- Portable across operating systems: Linux, macOS, Windows are supported
- Alpaka directly uses vendor API’s. For example: Alpaka cuda backend uses Cuda API directly etc.

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

alpaka in the wild – example use case

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

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

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

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

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

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

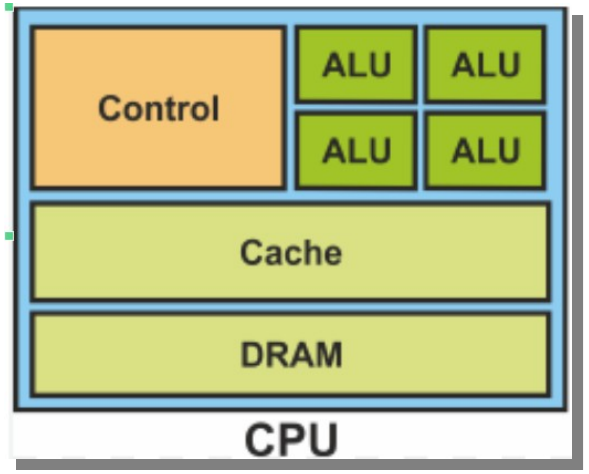
- Full source code
- Issue tracker
- Installation instructions
- Small examples

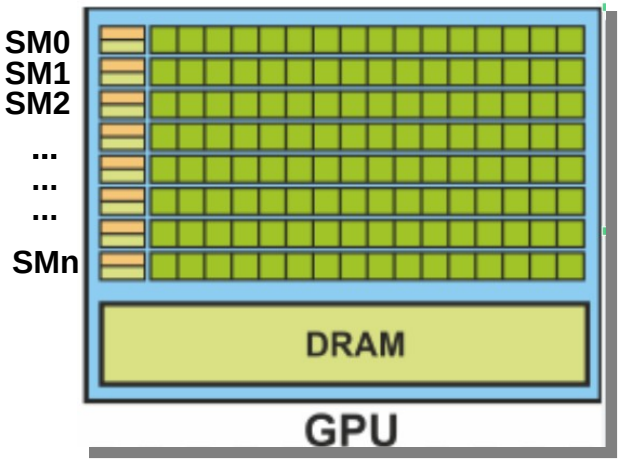
alpaka in a Nutshell

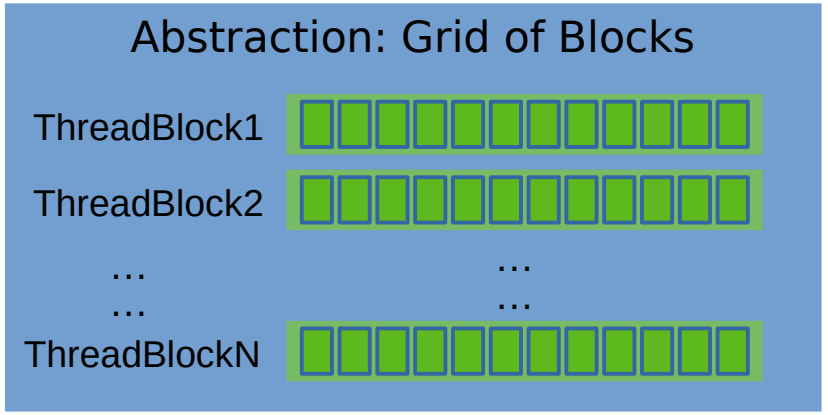
Main difference between CPU and GPU programming?

- CPU:
  - 1 thread per core and not may cores
  - Big cache size per thread
  - Low memory bandwidth(BW)

- GPU:
  - Massively parallel many threads per core (Streaming Multiprocessor) and many cores
  - Low cache size per thread
  - High memory BW

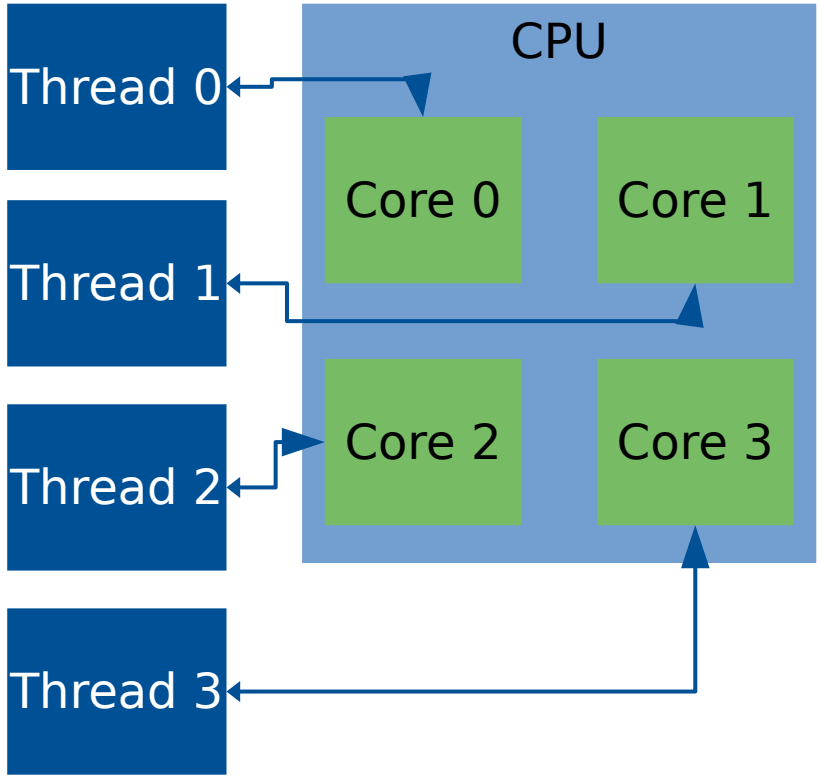






Basics: Thread mapping on CPUs

- CPU consists of multiple cores
  - Because of simultaneous multithreading there can be more logical than physical cores!
- alpaka Threads are executed by CPU cores.
- Single thread per block. Single block per core.
- Multiple elements per thread.

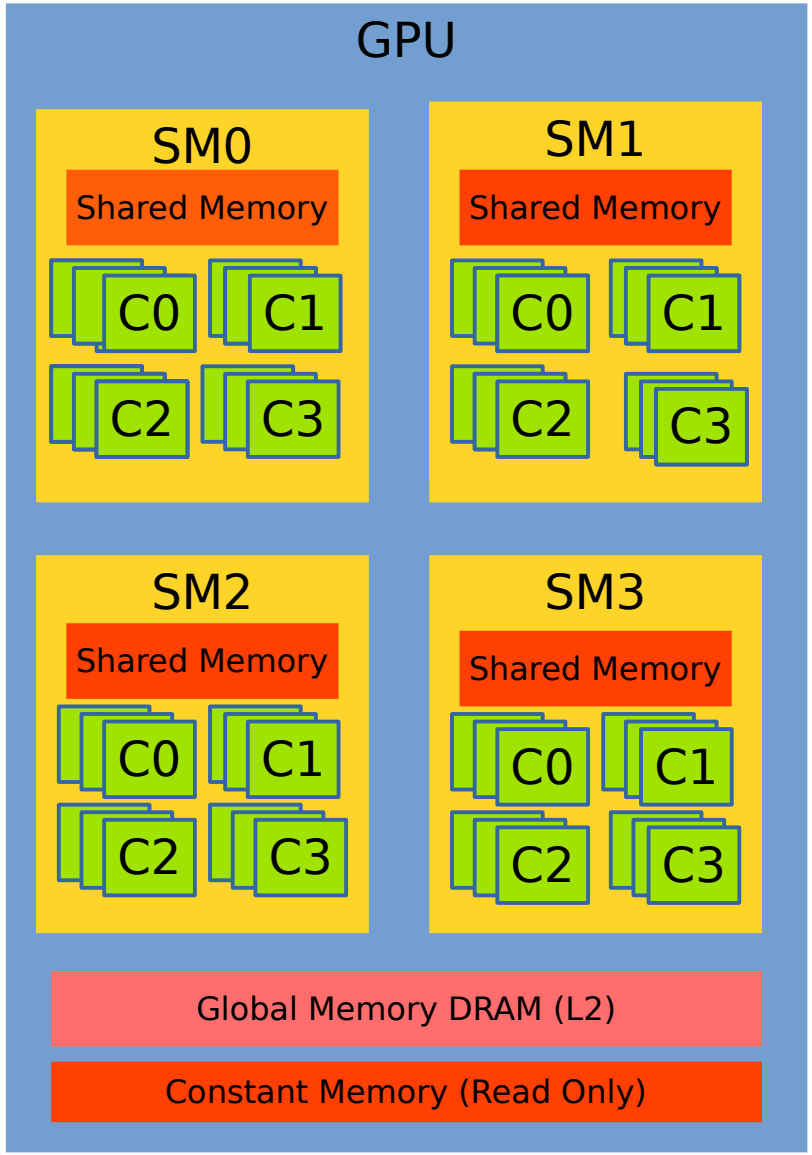


alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Thread Mapping on GPUs

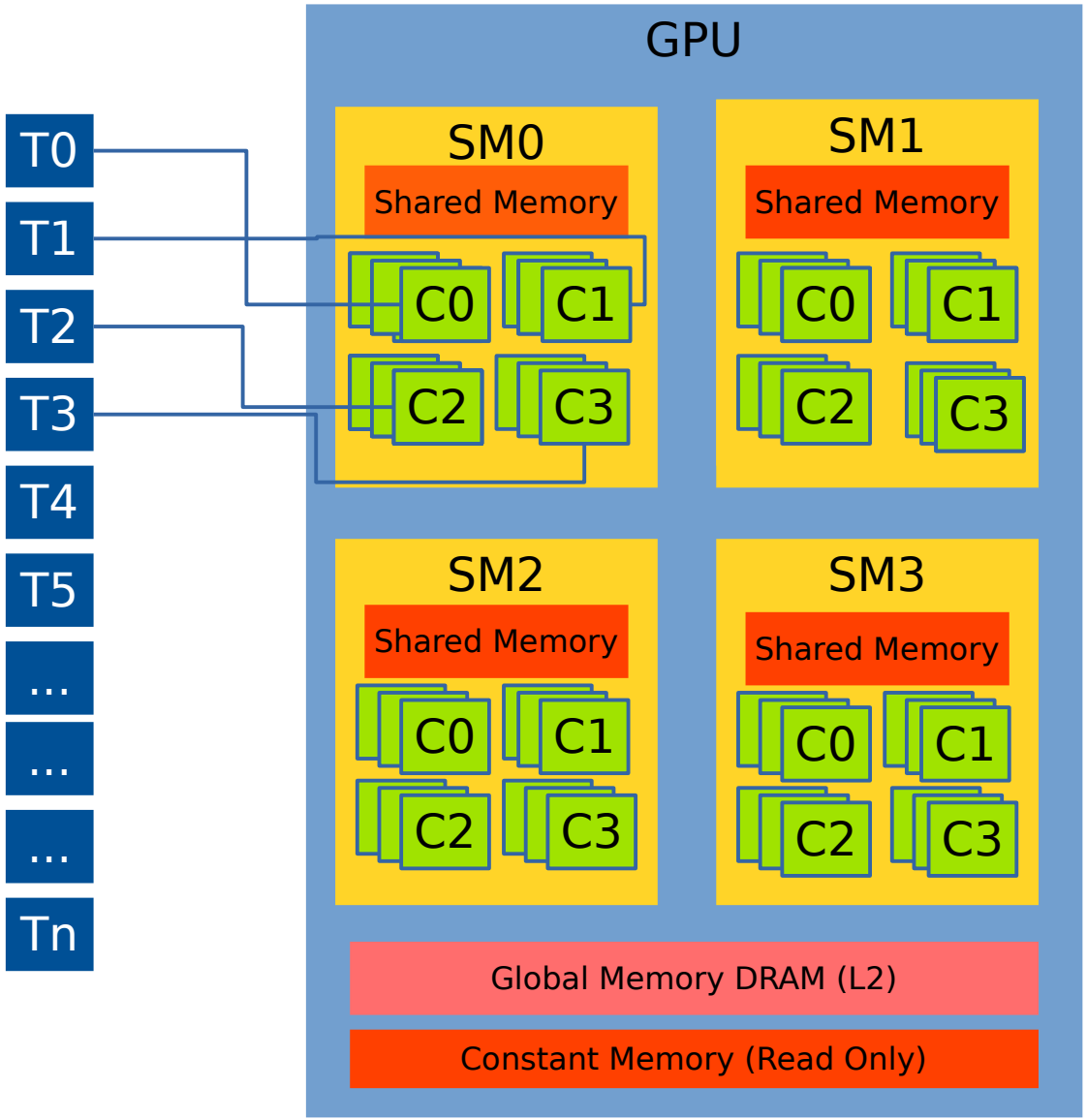
- GPU consists of **streaming multiprocessors** (SMs)
- Each SM consists of multiple **cores**
- A block has many threads and threads in a block are executed synchronously in the same SM. (SM can run more than one block)
- Each thread is matched to a core.



alpaka in a Nutshell

How distribute threads between SMs while matching each thread to a core?

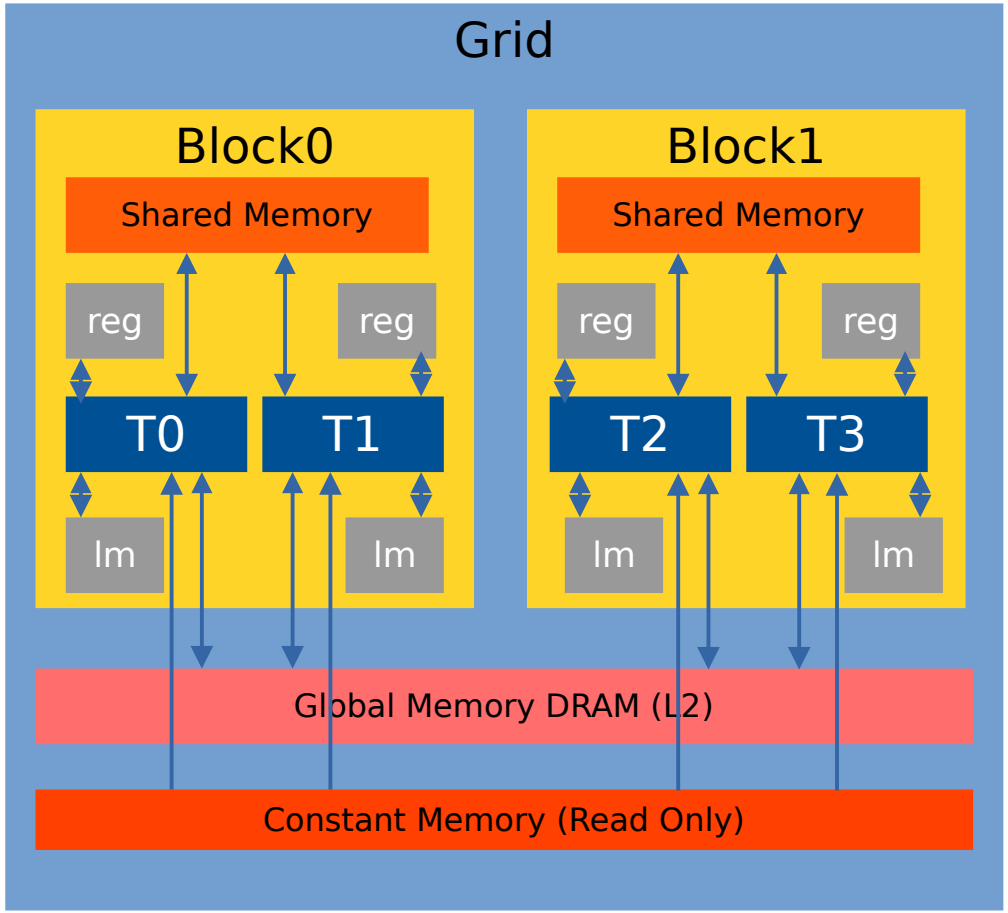
- Main determinants of mapping threads to the SMs:
  - Number of cores SM,
  - Register and local memory size for each thread,
  - Shared memory usage of each thread,
  - Limits for Threads per SM, Threads per Block, Blocks per Device
- Memory latencies: Global Memory and Constant memory has different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



alpaka in a Nutshell

Alpaka proposes and validates WorkDivision!

- WorkDivision data structure consists:
  - Number of Blocks per grid
  - Number of Threads per block
  - Elements per thread
- Alpaka validates and proposes correct parallelisation strategies to map threads to SMs



alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Easy definition, validation of Type of Parallelism by WorkDivision

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

```
// 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 */);
```

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Alpaka Features [Todo:Explain shortly?]

- **Buffer:** Not only a pointer; a data structure knows the device, extent, pitch
- **Device:** All functions depending on a device require it to be given as a parameter. Hence, device utilisation is transparent.
- **Queues:** Non-blocking queues as well as blocking queues are provided for all devices. Changes to the synchronicity of multiple tasks can be made on a per queue basis by changing the queue type at the place of creation. No need pair of functions like cudaMemcpyAsync and cudaMemcpy.
- Constant and Shared Memory Support
- https://alpaka.readthedocs.io/en/latest/advanced/rationale.html

alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Example: 1D Convolution Filter

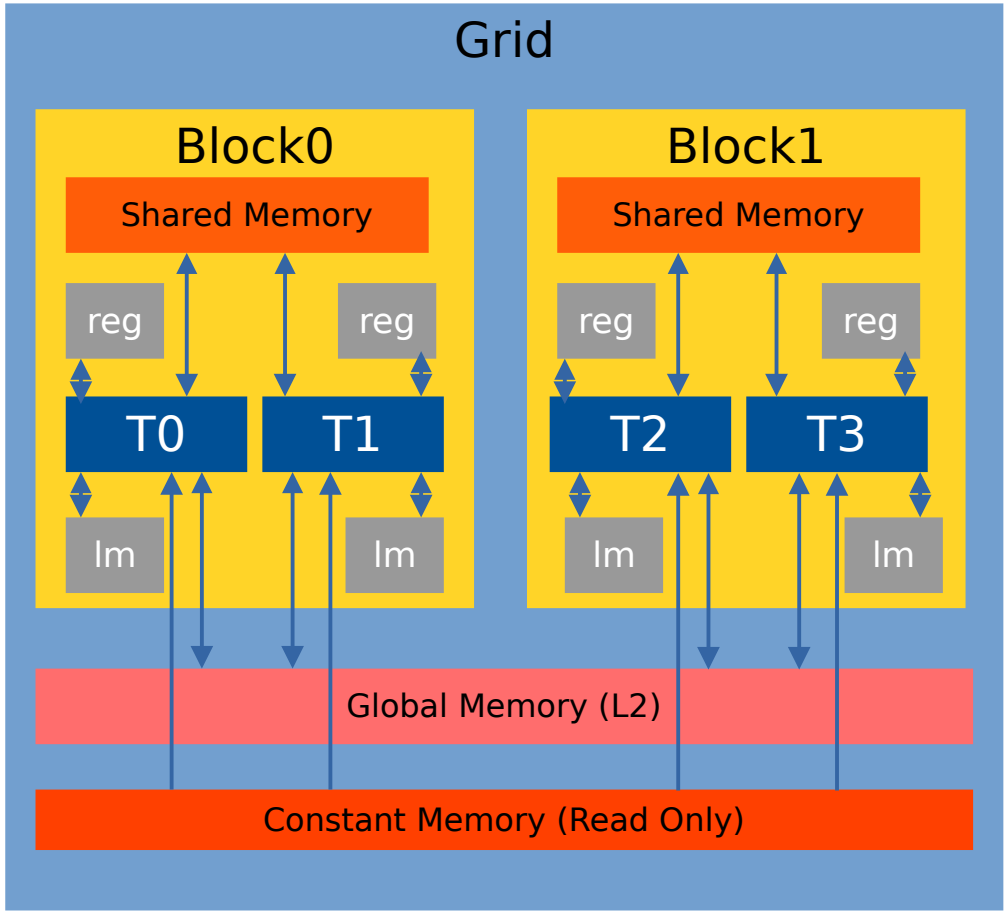
```
using DevAcc = alpaka::ExampleDefaultAcc<Dim, Idx>;
using QueueProperty = alpaka::Blocking;
using QueueAcc = alpaka::Queue<DevAcc, QueueProperty>;
using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
auto const platformHost = alpaka::PlatformCpu{};
auto const devHost = alpaka::getDevByIdx(platformHost, 0);

// Select a device
auto const platformAcc = alpaka::Platform<DevAcc>{};
auto const devAcc = alpaka::getDevByIdx(platformAcc, 0);
// Create a queue on the device
QueueAcc queue(devAcc);

// Allocate memory host input
auto hostInputMemory = alpaka::allocBuf<DataType, Idx>(devHost, inputSize);
// Fill array with data
for(size_t i = 0; i < inputSize; i++)
    hostInputMemory[i] = static_cast<DataType>(i + 1);

// Allocate memory host filter
auto hostFilterMemory = alpaka::allocBuf<DataType, Idx>(devHost, filterSize);
// Fill array with any data
for(size_t i = 0; i < filterSize; i++)
    hostFilterMemory[i] = static_cast<DataType>(i + 1) / 10.0f;
// Allocate memory in device
BufAcc inputDeviceMemory = alpaka::allocBuf<DataType, Idx>(devAcc, inputSize);
BufAcc filterDeviceMemory = alpaka::allocBuf<DataType, Idx>(devAcc, filterSize);
BufAcc outputDeviceMemory = alpaka::allocBuf<DataType, Idx>(devAcc, static_cast<Idx>(inputSize));

// Copy input and filter (convolution kernel array) from host to device
alpaka::memcpy(queue, inputDeviceMemory, hostInputMemory, inputSize);
alpaka::memcpy(queue, filterDeviceMemory, hostFilterMemory, filterSize);
// Make sure memcpy finished.
alpaka::wait(queue);
```

```
using Vec = alpaka::Vec<Dim, Idx>;
using WorkDiv = alpaka::WorkDivMembers<Dim, Idx>;

auto const elementsPerThread = Vec::all(static_cast<Idx>(1));
// Grid size
auto const threadsPerGrid = inputSize;
WorkDiv const workDiv = alpaka::getValidWorkDiv<DevAcc>(
    DevAcc, threadsPerGrid, elementsPerThread,false,
    alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

// Instantiate the kernel (paralelizable code) function-object
ConvolutionKernel convolutionKernel;

// Native pointers needed for the kernel execution function
DataType* nativeFilterDeviceMemory = alpaka::getPtrNative(filterDeviceMemory);
DataType* nativeInputDeviceMemory = alpaka::getPtrNative(inputDeviceMemory);
DataType* nativeOutputDeviceMemory = alpaka::getPtrNative(outputDeviceMemory);

// Run the kernel
alpaka::exec<DevAcc>( queue, workDiv,convolutionKernel, nativeInputDeviceMemory,
                      NativeFilterDeviceMemory, nativeOutputDeviceMemory,
                       InputSize, filterSize);

// Allocate memory on host
auto resultGpuHost = alpaka::allocBuf<DataType, Idx>(devHost, inputSize);
// Copy from device memory to host
alpaka::memcpy(queue, resultGpuHost, outputDeviceMemory, inputSize);
alpaka::wait(queue);
```

alpaka in a Nutshell

Alpaka proposes and validates WorkDivision!

- WorkDivision data structure consists:
  - Number of Blocks per grid
  - Number of Threads per block
  - Elements per thread
- Alpaka validates and proposes correct parallelisation strategies to map threads to SMs



alpaka in a Nutshell

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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 );
```

alpaka In a nutshell

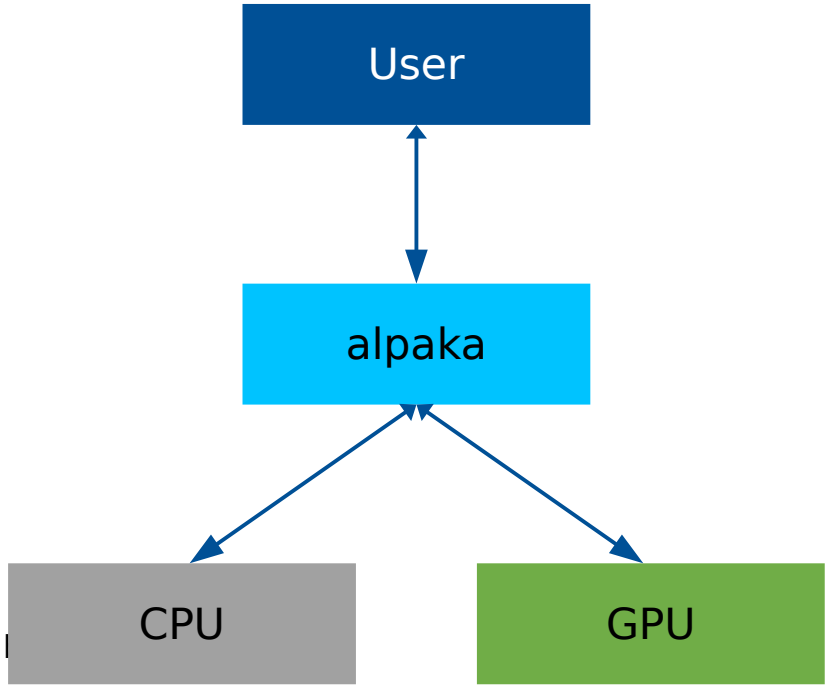
As a summary

Without alpaka

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

alpaka: one API to rule them all

- **Abstraction** (not hiding!) of the underlying hardware & software platforms
  - AMD, Nvidia, Intel GPUs, CPU
- **Easy change of the backend**
  - Code needs only minor adjustments to support different accelerators
- **Easy indexing of threads in kernels**
- **Easy setup of the type of parallelism by WorkDivision** (Block sizes in grid, Thread sizes in
- **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**.