



alpaka Documentation

Release 1.0.0-rc1

The alpaka Community

Jan 18, 2024

BASIC

1	Introduction	3
1.1	About alpaka	3
1.1.1	alpaka is	3
1.1.2	alpaka does not	4
2	Installation	5
2.1	Tests and Examples	5
3	Code Example	7
3.1	Use alpaka in your project	10
3.1.1	Use alpaka via <code>find_package</code>	10
3.1.2	Use alpaka via <code>add_subdirectory</code>	11
4	Abstraction	13
4.1	Task Parallelism	13
4.2	Data Parallelism	14
4.2.1	Thread	15
4.2.2	Block	16
4.2.3	Warp	17
4.2.4	Element	18
4.3	Summary	20
5	Library Interface	21
5.1	Structure	21
5.2	Interface Usage	22
5.2.1	Accelerator Functions	22
5.2.2	Kernel Definition	23
5.2.3	Index and Work Division	24
5.2.4	Memory fences	24
5.2.5	Memory Management	25
5.2.6	Kernel Execution	25
6	Cheatsheet	27
6.1	General	27
6.2	Accelerator, Platform and Device	27
6.3	Queue and Events	28
6.4	Memory	28
6.5	Kernel Execution	29
6.6	Kernel Implementation	30
7	Rationale	33
7.1	Interface Distinction	33
7.1.1	No Current Device:	33
7.1.2	No Default Device:	33
7.1.3	No Default Queue:	34

7.2	No Implicit Built-in Variables and Functions:	34
7.3	No Language Extensions:	34
7.4	No Dimensionality Restriction:	34
7.5	Integral Sizes of Arbitrary Type:	35
7.6	No Synchronous (Blocking) and Asynchronous (Non-Blocking) Function Versions:	35
7.7	Memory Management	35
7.8	Acceleratable Functions	35
7.9	Execution Domain Specifications	37
7.10	Kernel Function	37
7.10.1	Requirements	37
7.10.2	Implementation Variants	37
7.10.3	Implementation Notes	38
7.10.4	Access to Accelerator-Dependent Functionality	38
7.11	Index and Work Division	39
7.12	Block Shared Memory	39
7.12.1	Static Block Shared Memory	39
7.12.2	Dynamic Block Shared Memory	39
8	Mapping onto Specific Hardware Architectures	41
8.1	x86 CPUs	41
8.1.1	Thread	42
8.1.2	Warp	42
8.1.3	Block	42
8.1.4	Threading Mechanisms	43
8.2	GPUs (CUDA/HIP)	44
9	CMake Arguments	45
9.1	CMake Presets	45
9.1.1	Modifying and Extending Presets	45
9.2	Arguments	46
9.2.1	Common	47
9.2.2	CPU Serial	48
9.2.3	C++ Threads	48
9.2.4	Intel TBB	48
9.2.5	OpenMP 2 Grid Block	48
9.2.6	OpenMP 2 Block thread	49
9.2.7	CUDA	49
9.2.8	HIP	50
9.2.9	SYCL	50
10	Compiler Specifics	51
10.1	Choosing the correct Standard Library in Clang	51
10.1.1	Choose a specific libstdc++ version	51
10.1.2	Selecting libc++	52
11	Similar Projects	53
11.1	KOKKOS	53
11.2	Thrust	53
12	Back-ends	55
12.1	Accelerator Implementations	55
12.2	Serial	57
12.3	Threads	57
12.3.1	Execution	57
12.4	OpenMP	57
12.4.1	Execution	57
12.4.2	Index	57
12.4.3	Atomic	57
12.5	CUDA	58

12.5.1	CUDA Runtime API	60
12.6	HIP	63
12.6.1	Current Restrictions on HCC platform	63
12.6.2	Compiling HIP from Source	64
12.6.3	Verifying HIP Installation	65
12.6.4	Compiling Examples with HIP Back End	65
12.6.5	Random Number Generator Library rocRAND for HIP Back End	65
13	Details	67
13.1	Concept Implementations	68
13.2	Template Specialization Selection on Arbitrary Conditions	71
13.3	Argument dependent lookup for math functions	72
14	Coding Guidelines	73
14.1	General	73
14.2	Naming	73
14.3	Types	73
14.4	Type Qualifiers	74
14.5	Variables	74
14.6	Comments	74
14.7	Functions	74
14.8	Templates	75
14.9	Traits	75
14.10	Includes	75
15	Sphinx	77
15.1	Build Locally	77
15.2	readthedocs	78
15.3	Useful Links	78
16	Automatic Testing	79
16.1	GitHub Actions	79
16.1.1	clang-format	79
16.2	GitLab CI	79
16.2.1	The Container Registry	81
16.2.2	The Job Generator	81
16.2.3	Custom jobs	84
17	Indices and Tables	85
	Index	87



alpaka - An Abstraction Library for Parallel Kernel Acceleration

The alpaka library is a header-only C++17 abstraction library for accelerator development. Its aim is to provide performance portability across accelerators through the abstraction (not hiding!) of the underlying levels of parallelism.

Caution: The readthedocs pages are provided with best effort, but may contain outdated sections.

Generally, **follow the manual pages in-order** to get started. Individual chapters are based on the information of the chapters before.

Note: Are you looking for our latest Doxygen docs for the API?

- See <https://alpaka-group.github.io/alpaka/>
-

INTRODUCTION

The *alpaka* library defines and implements an abstract interface for the *hierarchical redundant parallelism* model. This model exploits task- and data-parallelism as well as memory hierarchies at all levels of current multi-core architectures. This allows to achieve performance portability across various types of accelerators by ignoring specific unsupported levels and utilizing only the ones supported on a specific accelerator. All hardware types (CPUs, GPUs and other accelerators) are treated and can be programmed in the same way. The *alpaka* library provides back-ends for *CUDA*, *OpenMP*, *HIP*, *SYCL* and other technologies. The trait-based C++ template interface provided allows for straightforward user-defined extension of the library to support other accelerators.

The library name *alpaka* is an acronym standing for **A**bstraction **L**ibrary for **P**arallel **K**ernel **A**cceleration.

1.1 About alpaka

1.1.1 alpaka is ...

Abstract

It describes parallel execution on multiple hierarchy levels. It allows to implement a mapping to various hardware architectures but is no optimal mapping itself.

Sustainable

alpaka decouples the application from the availability of different accelerator frameworks in different versions, such as OpenMP, CUDA, HIP, etc. (50% on the way to reach full performance portability).

Heterogeneous

An identical algorithm / kernel can be executed on heterogeneous parallel systems by selecting the target device. This allows the best performance for each algorithm and/or a good utilization of the system without major code changes.

Maintainable

alpaka allows to provide a single version of the algorithm / kernel that can be used by all back-ends. There is no need for “copy and paste” kernels with different API calls for different accelerators. All the accelerator dependent implementation details are hidden within the *alpaka* library.

Testable

Due to the easy back-end switch, no special hardware is required for testing the kernels. Even if the simulation itself always uses the *CUDA* back-end, the tests can completely run on a CPU. As long as the *alpaka* library is thoroughly tested for compatibility between the acceleration back-ends, the user simulation code is guaranteed to generate identical results (ignoring rounding errors / non-determinism) and is portable without any changes.

Optimizable

Everything in *alpaka* can be replaced by user code to optimize for special use-cases.

Extensible

Every concept described by the *alpaka* abstraction can be implemented by users. Therefore it is possible to non-intrusively define new devices, queues, buffer types or even whole accelerator back-ends.

Data Structure Agnostic

The user can use and define arbitrary data structures.

1.1.2 alpaka does not ...

Automatically provide an optimal mapping of kernels to various acceleration platforms

Except in trivial examples an optimal execution always depends on suitable selected data structures. An adaptive selection of data structures is a separate topic that has to be implemented in a distinct library.

Automatically optimize concurrent data access

alpaka does not provide feature to create optimized memory layouts.

Handle differences in arithmetic operations

For example, due to **different rounding** or different implementations of floating point operations, results can differ slightly between accelerators.

Guarantee determinism of results

Due to the freedom of the library to reorder or repartition the threads within the tasks it is not possible or even desired to preserve deterministic results. For example, the non-associativity of floating point operations give non-deterministic results within and across accelerators.

The *alpaka* library is aimed at parallelization on shared memory, i.e. within nodes of a cluster. It does not compete with libraries for distribution of processes across nodes and communication among those. For these purposes libraries like MPI (Message Passing Interface) or others should be used. MPI is situated one layer higher and can be combined with *alpaka* to facilitate the hardware of a whole heterogeneous cluster. The *alpaka* library can be used for parallelization within nodes, MPI for parallelization across nodes.

INSTALLATION

```
# Clone alpaka from github.com
git clone --branch 0.9.0 https://github.com/alpaka-group/alpaka.git
cd alpaka
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/install/ ..
cmake --install .
```

2.1 Tests and Examples

Build and run examples:

```
# ..
cmake -Dalpaka_BUILD_EXAMPLES=ON ..
cmake --build . -t vectorAdd
./example/vectorAdd/vectorAdd # execution
```

Build and run tests:

```
# ..
cmake -DBUILD_TESTING=ON ..
cmake --build .
ctest
```

Enable accelerators:

Alpaka uses different accelerators to execute kernels on different processors. To use a specific accelerator in alpaka, two steps are required.

1. Enable the accelerator during the CMake configuration time of the project.
2. Select a specific accelerator in the source code.

By default, no accelerator is enabled because some combinations of compilers and accelerators do not work, see the table of [supported compilers](#). To enable an accelerator, you must set a CMake flag via `cmake .. -Dalpaka_ACC_<acc>_ENABLE=ON` when you create a new build. The following example shows how to enable the CUDA accelerator and build an alpaka project:

```
cmake -Dalpaka_ACC_GPU_CUDA_ENABLE=ON ...
```

In the overview of [cmake arguments](#) you will find all CMake flags for activating the different accelerators. How to select an accelerator in the source code is described on the [example page](#).

Warning: If an accelerator is selected in the source code that is not activated during CMake configuration time, a compiler error occurs.

Hint: When the test or examples are activated, the alpaka build system automatically activates the `serial` backend, as it is needed for many tests. Therefore, the tests are run with the `serial` backend by default. If you want to test another backend, you have to activate it at CMake configuration time, for example the HIP backend: `cmake .. -DBUILD_TESTING=ON -Dalpaka_ACC_GPU_HIP_ENABLE=ON`. Some alpaka tests use a selector algorithm to choose a specific accelerator for the test cases. The selector works with accelerator priorities. Therefore, it is recommended to enable only one accelerator for a build to make sure that the right one is used.

CODE EXAMPLE

The following example shows a small hello word example written with alpaka that can be run on different processors.

Listing 1: helloWorld.cpp

```
/* Copyright 2023 Benjamin Worpitz, Erik Zenker, Bernhard Manfred Gruber, Jan Stephan
 * SPDX-License-Identifier: ISC
 */

#include <alpaka/alpaka.hpp>
#include <alpaka/example/ExampleDefaultAcc.hpp>

#include <iostream>

//! Hello World Kernel
//!
//! Prints "[x, y, z][gtid] Hello World" where tid is the global thread number.
struct HelloWorldKernel
{
    template<typename TAcc>
    ALPAKA_FN_ACC auto operator()(TAcc const& acc) const -> void
    {
        using Dim = alpaka::Dim<TAcc>;
        using Idx = alpaka::Idx<TAcc>;
        using Vec = alpaka::Vec<Dim, Idx>;
        using Vec1 = alpaka::Vec<alpaka::DimInt<1u>, Idx>;

        // In the most cases the parallel work distribution depends
        // on the current index of a thread and how many threads
        // exist overall. These information can be obtained by
        // getIdx() and getWorkDiv(). In this example these
        // values are obtained for a global scope.
        Vec const globalThreadId = alpaka::getIdx<alpaka::Grid, alpaka::Threads>
↳(acc);
        Vec const globalThreadExtent = alpaka::getWorkDiv<alpaka::Grid,
↳alpaka::Threads>(acc);

        // Map the three dimensional thread index into a
        // one dimensional thread index space. We call it
        // linearize the thread index.
        Vec1 const linearizedGlobalThreadId = alpaka::mapIdx<1u>(globalThreadId,
↳globalThreadExtent);

        // Each thread prints a hello world to the terminal
        // together with the global index of the thread in
```

(continues on next page)

(continued from previous page)

```

    // each dimension and the linearized global index.
    // Mind, that alpaka uses the mathematical index
    // order [z][y][x] where the last index is the fast one.
    printf(
        "[z:%u, y:%u, x:%u][linear:%u] Hello World\n",
        static_cast<unsigned>(globalThreadIdx[0u]),
        static_cast<unsigned>(globalThreadIdx[1u]),
        static_cast<unsigned>(globalThreadIdx[2u]),
        static_cast<unsigned>(linearizedGlobalThreadIdx[0u]));
    }
};

auto main() -> int
{
    // Fallback for the CI with disabled sequential backend
    #if defined(ALPAKA_CI) && !defined(ALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED)
        return EXIT_SUCCESS;
    #else
        // Define the index domain
        //
        // Depending on your type of problem, you have to define
        // the dimensionality as well as the type used for indices.
        // For small index domains 16 or 32 bit indices may be enough
        // and may be faster to calculate depending on the accelerator.
        using Dim = alpaka::DimInt<3>;
        using Idx = std::size_t;

        // Define the accelerator
        //
        // It is possible to choose from a set of accelerators:
        // - AccGpuCudaRt
        // - AccGpuHipRt
        // - AccCpuThreads
        // - AccCpuOmp2Threads
        // - AccCpuOmp2Blocks
        // - AccCpuTbbBlocks
        // - AccCpuSerial
        //
        // Each accelerator has strengths and weaknesses. Therefore,
        // they need to be chosen carefully depending on the actual
        // use case. Furthermore, some accelerators only support a
        // particular workdiv, but workdiv can also be generated
        // automatically.

        // By exchanging the Acc and Queue types you can select where to execute the
        ↪ kernel.
        // using Acc = alpaka::AccCpuSerial<Dim, Idx>;
        using Acc = alpaka::ExampleDefaultAcc<Dim, Idx>;
        std::cout << "Using alpaka accelerator: " << alpaka::getAccName<Acc>() <<
        ↪ std::endl;

        // Defines the synchronization behavior of a queue
        //
        // choose between Blocking and NonBlocking
        using QueueProperty = alpaka::Blocking;
        using Queue = alpaka::Queue<Acc, QueueProperty>;

```

(continues on next page)

(continued from previous page)

```

// Select a device
//
// The accelerator only defines how something should be
// parallelized, but a device is the real entity which will
// run the parallel program. The device can be chosen
// by id (0 to the number of devices minus 1) or you
// can also retrieve all devices in a vector (getDevs()).
// In this example the first devices is chosen.
auto const platformAcc = alpaka::Platform<Acc>{};
auto const devAcc = alpaka::getDevByIdx(platformAcc, 0);

// Create a queue on the device
//
// A queue can be interpreted as the work queue
// of a particular device. Queues are filled with
// tasks and alpaka takes care that these
// tasks will be executed. Queues are provided in
// non-blocking and blocking variants.
// The example queue is a blocking queue to a cpu device,
// but it also exists as non-blocking queue for this
// device (QueueCpuNonBlocking).
Queue queue(devAcc);

// Define the work division
//
// A kernel is executed for each element of a
// n-dimensional grid distinguished by the element indices.
// The work division defines the number of kernel instantiations as
// well as the type of parallelism used by the kernel execution task.
// Different accelerators have different requirements on the work
// division. For example, the sequential accelerator can not
// provide any thread level parallelism (synchronizable as well as non-
↪synchronizable),
// whereas the CUDA accelerator can spawn hundreds of synchronizing
// and non synchronizing threads at the same time.
//
// The workdiv is divided in three levels of parallelization:
// - grid-blocks:      The number of blocks in the grid (parallel, not-
↪synchronizable)
// - block-threads:    The number of threads per block (parallel, synchronizable).
//                      Each thread executes one kernel invocation.
// - thread-elements:  The number of elements per thread (sequential, not-
↪synchronizable).
//                      Each kernel has to execute its elements sequentially.
//
// - Grid      : consists of blocks
// - Block     : consists of threads
// - Elements  : consists of elements
//
// Threads in the same grid can access the same global memory,
// while threads in the same block can access the same shared
// memory. Elements are supposed to be used for vectorization.
// Thus, a thread can process data element size wise with its
// vector processing unit.
using Vec = alpaka::Vec<Dim, Idx>;

```

(continues on next page)

(continued from previous page)

```
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
//
// Kernels can be everything that is trivially copyable, has a
// callable operator() and takes the accelerator as first
// argument. So a kernel can be a class or struct, a lambda, etc.
HelloWorldKernel helloWorldKernel;

// Run the kernel
//
// To execute the kernel, you have to provide the
// work division as well as the additional kernel function
// parameters.
// The kernel execution task is enqueued into an accelerator queue.
// The queue can be blocking or non-blocking
// depending on the chosen queue type (see type definitions above).
// Here it is synchronous which means that the kernel is directly executed.
alpaka::exec<Acc>(
    queue,
    workDiv,
    helloWorldKernel
    /* put kernel arguments here */);
alpaka::wait(queue);

return EXIT_SUCCESS;
#endif
}
```

3.1 Use alpaka in your project

We recommend to use CMake for integrating alpaka into your own project. There are two possible methods.

3.1.1 Use alpaka via find_package

The `find_package` method requires alpaka to be *installed* in a location where CMake can find it.

Hint: If you do not install alpaka in a default path such as `/usr/local/` you have to set the CMake argument `-Dalpaka_ROOT=/path/to/alpaka/install`.

The following example shows a minimal example of a `CMakeLists.txt` that uses alpaka:

Listing 2: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.22)
project("myexample" CXX)

find_package(alpaka REQUIRED)
alpaka_add_executable(${PROJECT_NAME} helloWorld.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC alpaka::alpaka)
```

In the CMake configuration phase of the project, you must activate the accelerator you want to use:

```
cd <path/to/the/project/root>
mkdir build && cd build
cmake .. -Dalpaka_ACC_GPU_CUDA_ENABLE=ON
cmake --build .
./myexample
```

A complete list of CMake flags for the accelerator can be found [here](#).

If the configuration was successful and CMake found the CUDA SDK, the C++ template accelerator type `alpaka::AccGpuCudaRt` is available.

3.1.2 Use alpaka via add_subdirectory

The `add_subdirectory` method does not require alpaka to be installed. Instead, the alpaka project folder must be part of your project hierarchy. The following example expects alpaka to be found in the `project_path/thirdParty/alpaka`:

Listing 3: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.22)
project("myexample" CXX)

add_subdirectory(thirdParty/alpaka)
alpaka_add_executable(${PROJECT_NAME} helloWorld.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC alpaka::alpaka)
```

The CMake configure and build commands are the same as for the `find_package` approach.

ABSTRACTION

Note: Objective of the abstraction is to separate the parallelization strategy from the algorithm itself. Algorithm code written by users should not depend on any parallelization library or specific strategy. This would enable exchanging the parallelization back-end without any changes to the algorithm itself. Besides allowing to test different parallelization strategies this also makes it possible to port algorithms to new, yet unsupported, platforms.

Parallelism and memory hierarchies at all levels need to be exploited in order to achieve performance portability across various types of accelerators. Within this chapter an abstraction will be derived that tries to provide a maximum of parallelism while simultaneously considering implementability and applicability in hardware.

Looking at the current HPC hardware landscape, we often see nodes with multiple sockets/processors extended by accelerators like GPUs or Intel Xeon Phi, each with their own processing units. Within a CPU or a Intel Xeon Phi there are cores with hyper-threads, vector units and a large caching infrastructure. Within a GPU there are many small cores and only few caches. Each entity in the hierarchy has access to different memories. For example, each socket / processor manages its RAM, while the cores additionally have non-explicit access to L3, L2 and L1 caches. On a GPU there are global, constant, shared and other memory types which all can be accessed explicitly. The interface has to abstract from these differences without sacrificing speed on any platform.

A process running on a multi-socket node is the largest entity within *alpaka*. The abstraction is only about the task and data parallel execution on the process/node level and down. It does not provide any primitives for inter-node communication. However, such libraries can be combined with *alpaka*.

An application process always has a main thread and is by definition running on the host. It can access the host memory and various accelerator devices. Such accelerators can be GPUs, Intel Xeon Phis, the host itself or other devices. Thus, the host not necessarily has to be different from the accelerator device used for the computations. For instance, an Intel Xeon Phi simultaneously can be the host and the accelerator device.

The *alpaka* library can be used to offload the parallel execution of task and data parallel work simultaneously onto different accelerator devices.

4.1 Task Parallelism

One of the basic building blocks of modern applications is task parallelism. For example, the operating system scheduler, deciding which thread of which process gets how many processing time on which CPU core, enables task parallelism of applications. It controls the execution of different tasks on different processing units. Such task parallelism can be, for instance, the output of the progress in parallel to a download. This can be implemented via two threads executing two different tasks.

The valid dependencies between tasks within an application can be defined as a DAG (directed acyclic graph) in all cases. The tasks are represented by nodes and the dependencies by edges. In this model, a task is ready to be executed if the number of incoming edges is zero. After a task finished its work, it is removed from the graph as well as all of its outgoing edges. This reduces the number of incoming edges of subsequent tasks.

The problem with this model is the inherent overhead and the missing hardware and API support. When it is directly implemented as a graph, at least all depending tasks have to be updated and checked if they are ready to

be executed after a task finished. Depending on the size of the graph and the number of edges this can be a huge overhead.

OpenCL allows to define a task graph in a somewhat different way. Tasks can be enqueued into an out-of-order command queue combined with events that have to be finished before the newly enqueued task can be started. Tasks in the command queue with unmet dependencies are skipped and subsequent ones are executed. The `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` property of a command queue is an optional feature only supported by few vendors. Therefore, it can not be assumed to be available on all systems.

CUDA on the other hand does currently (version 7.5) not support such out-of-order queues in any way. The user has to define dependencies explicitly through the order the tasks are enqueued into the queues (called queues in *CUDA*). Within a queue, tasks are always executed in sequential order, while multiple queues are executed in parallel. Queues can wait for events enqueued into other queues.

In both APIs, *OpenCL* and *CUDA*, a task graph can be emulated by creating one queue per task and enqueueing a unique event after each task, which can be used to wait for the preceding task. However, this is not feasible due to the large queue and event creation costs as well as other overheads within this process.

Therefore, to be compatible with a wide range of APIs, the interface for task parallelism has to be constrained. Instead of a general DAG, multiple queues of sequentially executed tasks will be used to describe task parallelism. Events that can be enqueued into the queues enhance the basic task parallelism by enabling synchronization between different queues, devices or the host threads.

4.2 Data Parallelism

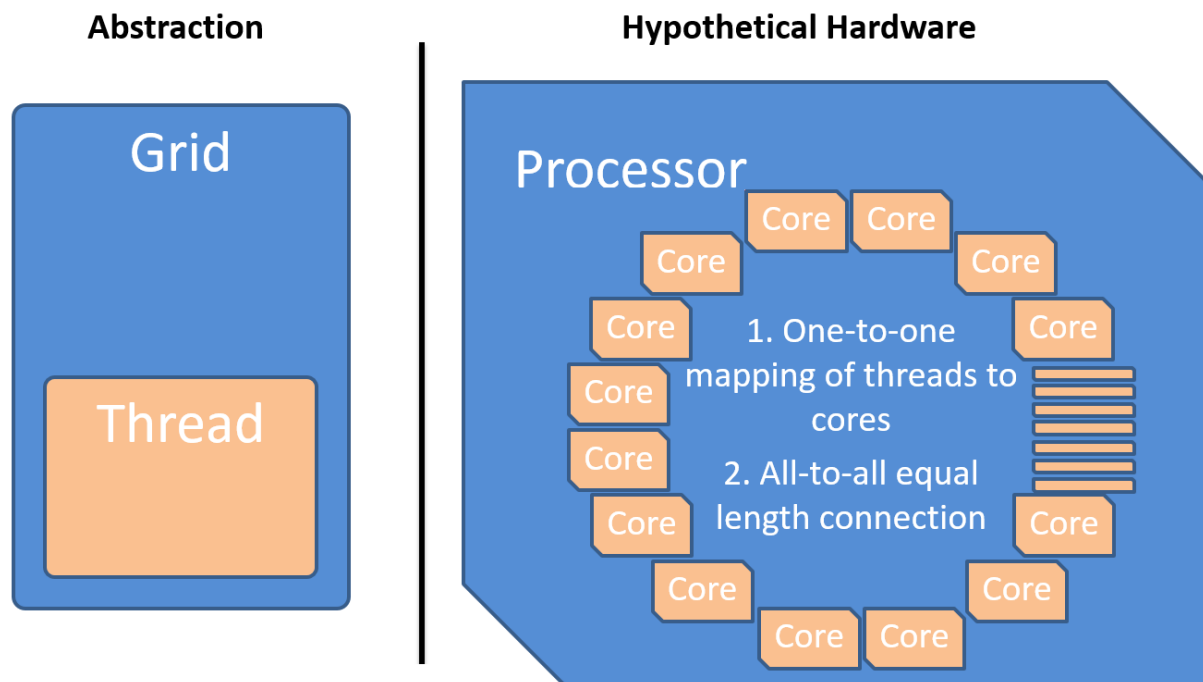
In contrast to task parallelism, data parallelism describes the execution of one and the same task on multiple, often related data elements. For example, an image color space conversion is a textbook example of a data parallel task. The same operation is executed independently on each pixel. Other data parallel algorithms additionally introduce dependencies between threads in the input-, intermediate-, or output-data. For example, the calculation of a brightness histogram has no input-data dependencies. However, all pixel brightness values finally have to be merged into a single result. Even these two simple examples show that it is necessary to think about the interaction of parallel entities to minimize the influence of data dependencies.

Furthermore, it is necessary to respect the principles of spatial and temporal locality. Current hardware is built around these locality principles to reduce latency by using hierarchical memory as a trade-off between speed and hardware size. Multiple levels of caches, from small and very fast ones to very large and slower ones exploit temporal locality by keeping recently referenced data as close to the actual processing units as possible. Spatial locality in the main memory is also important for caches because they are usually divided into multiple lines that can only be exchanged one cache line at a time. If one data element is loaded and cached, it is highly likely that nearby elements are also cached. If the pixels of an image are stored row wise but are read out column wise, the spatial locality assumption of many CPUs is violated and the performance suffers. GPUs on the other hand do not have a large caching hierarchy but allow explicit access to a fast memory shared across multiple cores. Therefore, the best way to process individual data elements of a data parallel task is dependent on the data structure as well as the underlying hardware.

The main part of the *alpaka* abstraction is the way it abstracts data parallelism and allows the algorithm writer to take into account the hierarchy of processing units, their data parallel features and corresponding memory regions. The abstraction developed is influenced and based on the groundbreaking *CUDA* and *OpenCL* abstractions of a multidimensional grid of threads with additional hierarchy levels in between. Another level of parallelism is added to those abstractions to unify the data parallel capabilities of modern hardware architectures. The explicit access to all hierarchy levels enables the user to write code that runs performant on all current platforms. However, the abstraction does not try to automatically optimize memory accesses or data structures but gives the user full freedom to use data structures matching the underlying hardware preferences.

4.2.1 Thread

Theoretically, a basic data parallel task can be executed optimally by executing one thread per independent data element. In this context, the term thread does not correspond to a native kernel-thread, an *OpenMP* thread, a *CUDA* thread, a user-level thread or any other such threading variant. It only represents the execution of a sequence of commands forming the desired algorithm on a per data element level. This ideal one-to-one mapping of data elements to threads leads to the execution of a multidimensional grid of threads corresponding to the data structure of the underlying problem. The uniform function executed by each of the threads is called a kernel. Some algorithms such as reductions require the possibility to synchronize or communicate between threads to calculate a correct result in a time optimal manner. Therefore our basic abstraction requires a n-dimensional grid of synchronizable threads each executing the same kernel. The following figure shows an hypothetical processing unit that could optimally execute this data parallel task. The threads are mapped one-to-one to the cores of the processor. For a time optimal execution, the cores have to have an all-to-all equal length connection for communication and synchronization.



The only difference between the threads is their positional index into the grid which allows each thread to compute a different part of the solution. Threads can always access their private registers and the global memory.

Registers

All variables with default scope within a kernel are automatically saved in registers and are not shared automatically. This memory is local to each thread and can not be accessed by other threads.

Global Memory

The global memory can be accessed from every thread in the grid as well as from the host thread. This is typically the largest but also the slowest memory available.

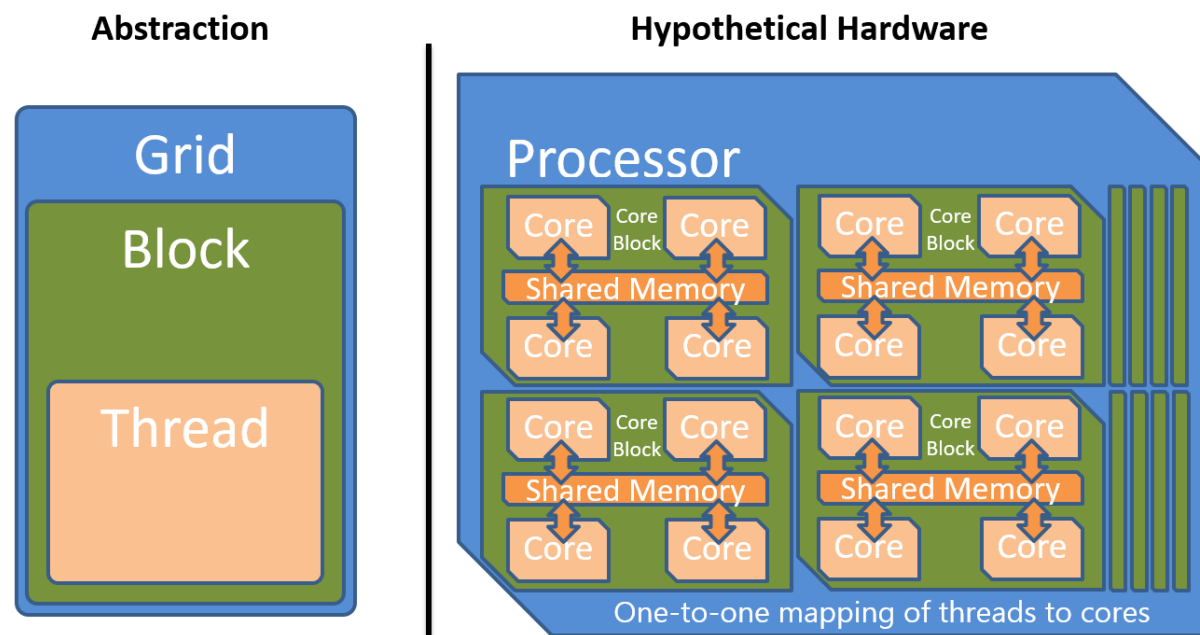
Individual threads within the grid are allowed to statically or dynamically allocate buffers in the global memory.

Prior to the execution of a task, the host thread copies the input buffers and allocates the output buffers onto the accelerator device. Pointers to these buffers then can be given as arguments to the task invocation. By using the index of each thread within the grid, the offset into the global input and output buffers can be calculated. After the computation has finished, the output buffer can be used either as input to a subsequent task or can be copied back to the host.

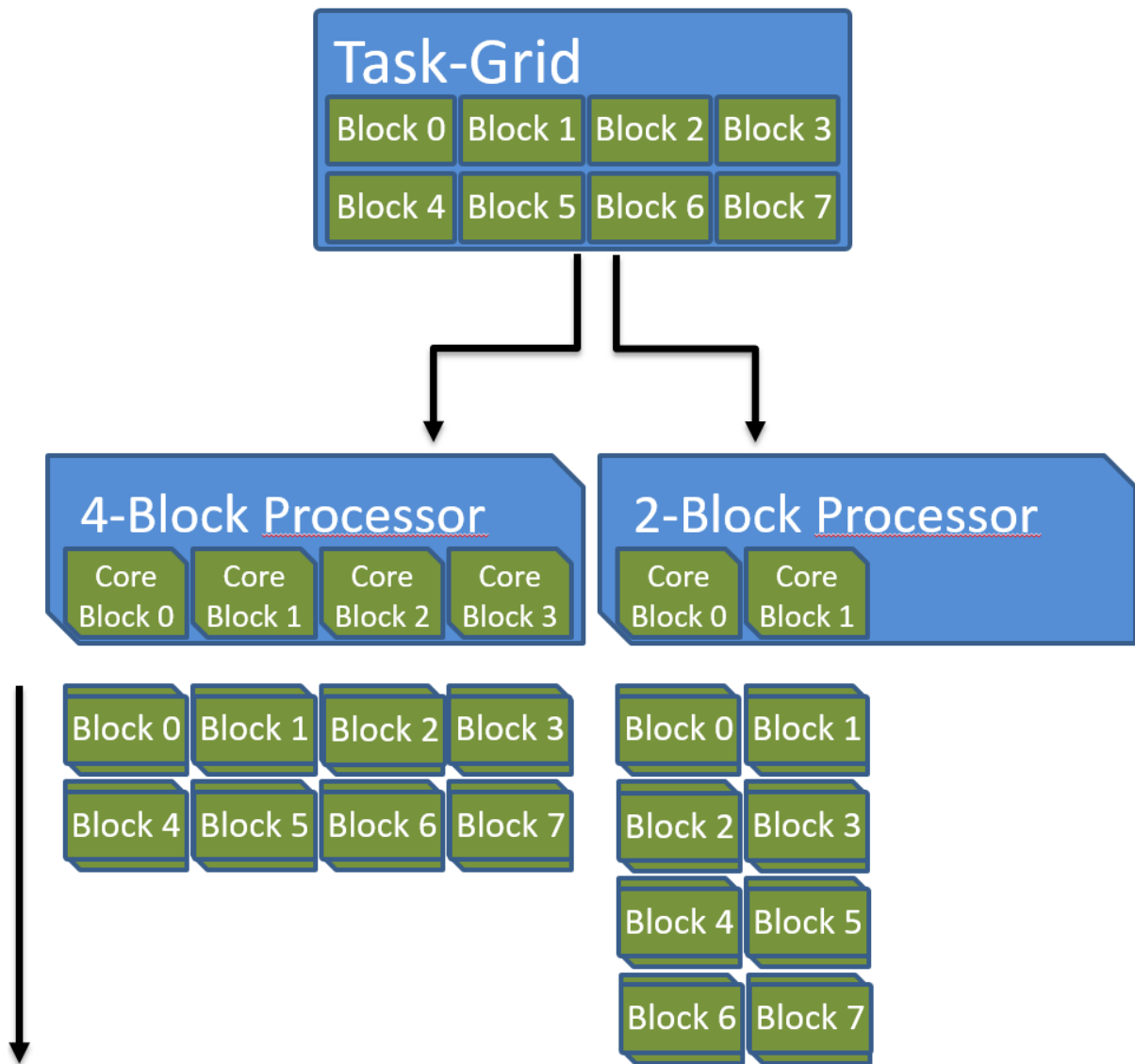
4.2.2 Block

Building a processor with possibly thousands of cores where all cores have an equal length connection for fast communication and synchronization is not viable. Either the processor size would have to grow exponentially with the number of cores or the all-to-all communication speed would decrease so much that computations on the processor would be impractical. Therefore, the communication and synchronization of threads has to be limited to sizes manageable by real hardware.

Figure [ref{fig:block}](#) depicts the solution of introducing a new hierarchy level in the abstraction. A hypothetical processor is allowed to provide synchronization and fast communication within blocks of threads but is not required to provide synchronization across blocks. The whole grid is subdivided into equal sized blocks with a fast but small shared memory. Current accelerator abstractions (*CUDA* and *OpenCL*) only support equal sized blocks. This restriction could possibly be lifted to support future accelerators with heterogeneous block sizes.



There is another reason why independent blocks are necessary. Threads that can communicate and synchronize require either a one-to-one mapping of threads to cores, which is impossible because the number of data elements is theoretically unlimited, or at least a space to store the state of each thread. Even old single core CPUs were able to execute many communicating and synchronizing threads by using cooperative or preemptive multitasking. Therefore, one might think that a single core would be enough to execute all the data parallel threads. But the problem is that even storing the set of registers and local data of all the possible millions of threads of a task grid is not always viable. The blocking scheme solves this by enabling fast interaction of threads on a local scale but additionally removes the necessity to store the state of all threads in the grid at once because only threads within a block must be executed in parallel. Within a block of cores there still has to be enough memory to store all registers of all contained threads. The independence of blocks allows applications to scale well across diverse devices. As can be seen in the following figure, the accelerator can assign blocks of the task grid to blocks of cores in arbitrary order depending on availability and workload.

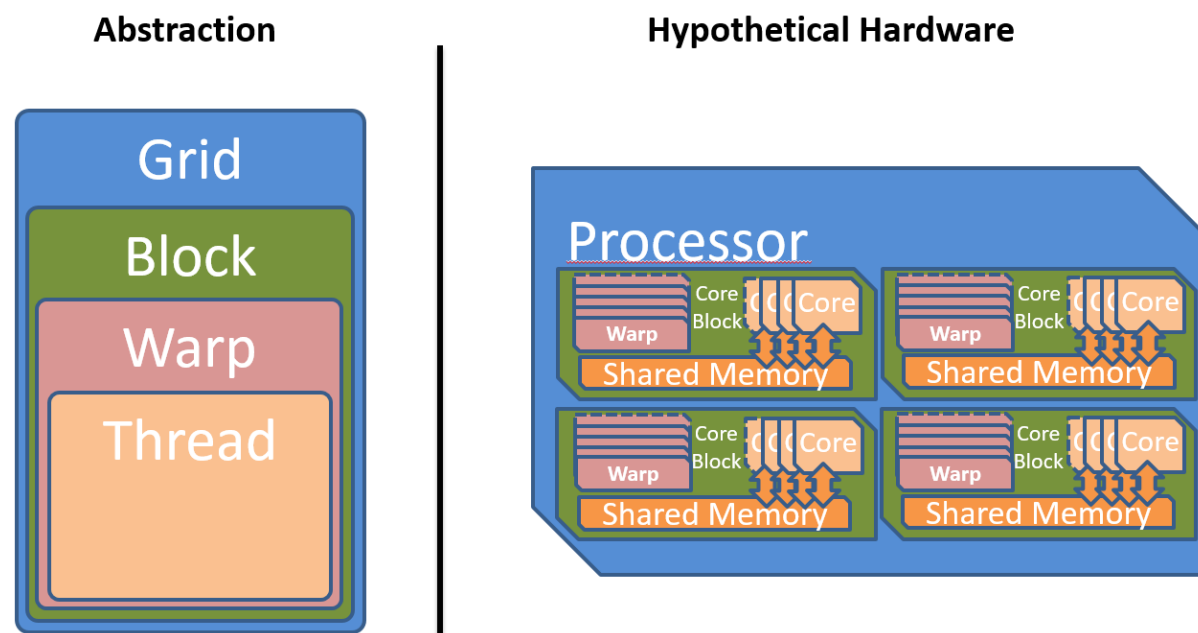


Shared Memory

Each block has its own shared memory. This memory can only be accessed explicitly by threads within the same block and gets discarded after the complete block finished its calculation. This memory is typically very fast but also very small. No variables are shared between kernels by default.

4.2.3 Warp

With the current abstraction only independent parallelism via blocks and synchronizable parallelism via threads can be expressed. However, there are more variants of parallelism in real hardware. Because all threads in the grid are executing the same kernel and even the same instruction at the same time when ignoring divergent control flows, a lot of chip space can be saved. Multiple threads can be executed in perfect synchronicity, which is also called lock-step. A group of such threads executing the same instruction at the same time is called a warp. All threads within a warp share a single instruction pointer (IP), and all cores executing the threads share one instruction fetch (IF) and instruction decode (ID) unit.



Even threads with divergent control flows can be executed within one warp. *CUDA*, for example, solves this by supporting predicated execution and warp voting. For long conditional branches the compiler inserts code which checks if all threads in the warp take the same branch. For small branches, where this is too expensive, all threads always execute all branches. Control flow statements result in a predicate and only in those threads where it is true, the predicated instructions will have an effect.

Not only *CUDA* GPUs support the execution of multiple threads in a warp. Full blown vector processors with good compilers are capable of combining multiple loop iterations containing complex control flow statements in a similar manner as *CUDA*.

Due to the synchronitiy of threads within a warp, memory operations will always occur at the same time in all threads. This allows to coalesce memory accesses. Different *CUDA* devices support different levels of memory coalescing. Older ones only supported combining multiple memory accesses if they were aligned and sequential in the order of thread indices. Newer ones support unaligned scattered accesses as long as they target the same 128 byte segment.

The ability of very fast context switches between warps and a queue of ready warps allows *CUDA* capable GPUs to hide the latency of global memory operations.

4.2.4 Element

To use the maximum available computing power of, for example, a modern x86 processor, the computation has to utilize the SIMD vector registers. Many current architectures support issuing a single instruction that can be applied to multiple data elements in parallel.

The original x86 instruction set architecture did not support SIMD instructions but has been enhanced with MMX (64 bit width registers), SSE (128 bit width registers), AVX (256 bit width registers) and AVX-512 (512 bit width registers) extensions. In varying degree, they allow to process multiple 32 bit and 64 bit floating point numbers as well as 8, 16, 32 and 64 bit signed and unsigned integers.

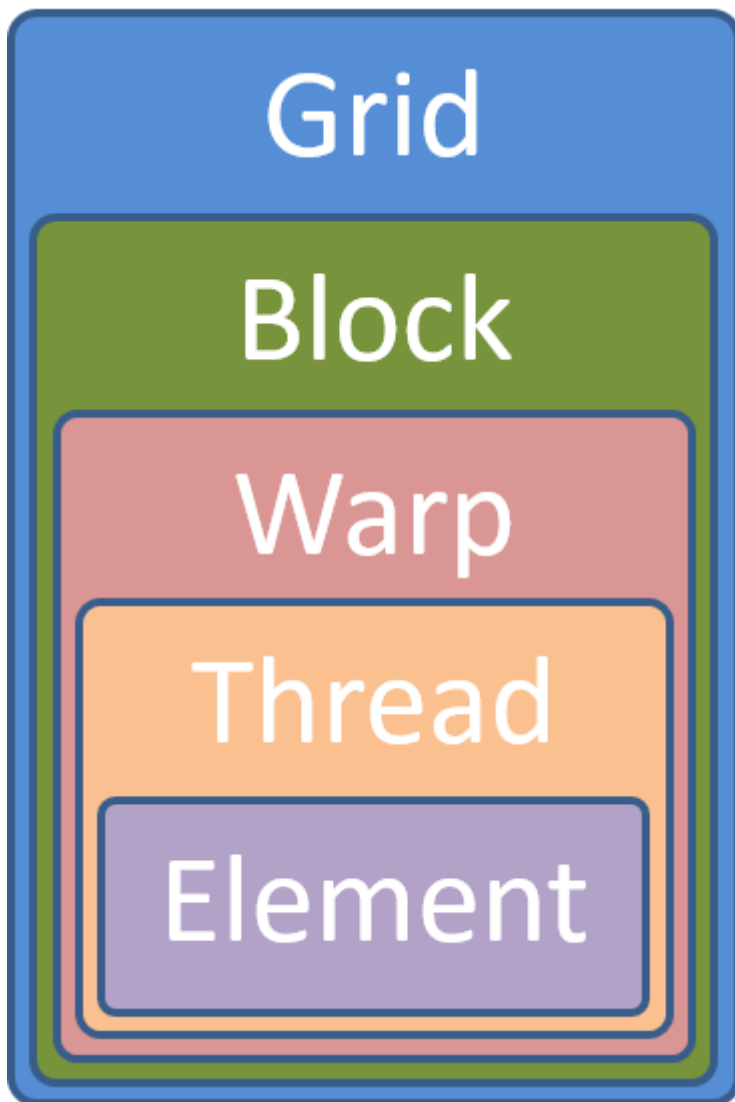
CUDA capable GPUs do not have vector registers where multiple values of type `float` or `double` can be manipulated by one instruction. Nevertheless, newer *CUDA* capable devices implement basic SIMD instructions on pairs of 16 bit values and quads of 8-bit values. They are described in the documentation of the [PTX instruction set architecture](#) chapter 9.7.13 but are only of any use in very special problem domains, for example for deep learning.

It would be optimal if the compiler could automatically vectorize our kernels when they are called in a loop and vectorization is supported by the underlying accelerator. However, besides full blown vector processors, mainstream CPUs do not support predicated execution or similar complex things within vector registers. At most, there is support for masking operations which allow to emulate at least some conditional branching. Therefore, this missing hardware capability has to be circumvented by the compiler. There are scientific research projects such as

the work done by Ralf Karrenberg et al [1, 2, 3] building on the *LLVM* compiler infrastructure supporting such whole-function vectorization. However, current mainstream compilers do not support automatic vectorization of basic, non trivial loops containing control flow statements (`if`, `else`, `for`, etc.) or other non-trivial memory operations. Therefore, it has to be made easier for the compiler to recognize the vectorization possibilities by making it more explicit.

The opposite of automatic whole function vectorization is the fully explicit vectorization of expressions via compiler intrinsics directly resulting in the desired assembly instruction. A big problem when trying to utilize fully explicit vectorization is, that there is no common foundation supported by all explicit vectorization methods. A wrapper unifying the x86 SIMD intrinsics found in the `intrin.h` or `x86intrin.h` headers with those supported on other platforms, for example ARM NEON (`arm_neon.h`), PowerPC AltiVec (`altivec.h`) or *CUDA* is not available and to write one is a huge task in itself. However, if this would become available in the future, it could easily be integrated into *alpaka* kernels.

Due to current compilers being unable to vectorize whole functions and the explicit vectorization intrinsics not being portable, one has to rely on the vectorization capabilities of current compilers for primitive loops only consisting of a few computations. By creating a grid of data elements, where multiple elements are processed per thread and threads are pooled in independent blocks, as it is shown in the figure below, the user is free to loop sequentially over the elements or to use vectorization for selected expressions within the kernel. Even the sequential processing of multiple elements per thread can be useful depending on the architecture. For example, the *NVIDIA cuBLAS* general matrix-matrix multiplication (GEMM) internally executes only one thread for each second matrix data element to better utilize the registers available per thread.



Note: The best solution to vectorization would be one, where the user does not have to do anything. This is not possible because the smallest unit supplied by the user is a kernel which is executed in threads which can synchronize.

It is not possible to execute multiple kernels sequentially to hide the vectorization by starting a kernel-thread for e.g. each 4th thread in a block and then looping over the 4 entries. This would prohibit the synchronization between these threads. By executing 4 fibers inside such a vectorization kernel-thread we would allow synchronization again but prevent the loop vectorizer from working.

4.3 Summary

This abstraction is called *Redundant Hierarchical Parallelism*. This term is inspired by the paper *The Future of Accelerator Programming: Abstraction, Performance or Can We Have Both?* [PDF](#) [DOI](#) It investigates a similar *concept of copious parallel programming* reaching 80%-90% of the native performance while comparing CPU and GPU centric versions of an *OpenCL* n-body simulation with a general version utilizing parallelism on multiple hierarchy levels.

The *CUDA* or *OpenCL* abstractions themselves are very similar to the one designed in the previous sections and consists of all but the Element level. However, as has been shown, all five abstraction hierarchy levels are necessary to fully utilize current architectures. By emulating unsupported or ignoring redundant levels of parallelism, algorithms written with this abstraction can always be mapped optimally to all supported accelerators. The following table summarizes the characteristics of the proposed hierarchy levels.

Hierarchy Level	Parallelism	Synchronizable
—	—	—
grid	sequential / parallel	– / X
block	parallel	–
warp	parallel	X
thread	parallel / lock-step	X
element	sequential	–

Depending on the queue a task is enqueued into, grids will either run in sequential order within the same queue or in parallel in different queues. They can be synchronized by using events. Blocks can not be synchronized and therefore can use the whole spectrum of parallelism ranging from fully parallel up to fully sequential execution depending on the device. Warps combine the execution of multiple threads in lock-step and can be synchronized implicitly by synchronizing the threads they contain. Threads within a block are executed in parallel warps and each thread computes a number of data elements sequentially.

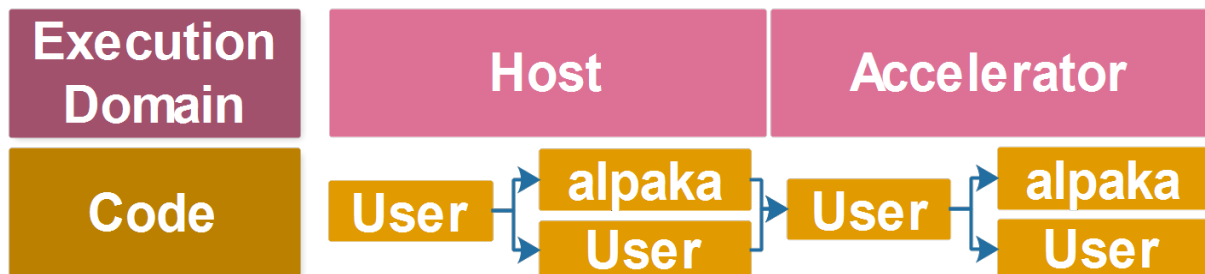
LIBRARY INTERFACE

As described in the chapter about the *Abstraction*, the general design of the library is very similar to *CUDA* and *OpenCL* but extends both by some points, while not requiring any language extensions. General interface design as well as interface implementation decisions differentiating *alpaka* from those libraries are described in the Rationale section. It uses C++ because it is one of the most performant languages available on nearly all systems. Furthermore, C++17 allows to describe the concepts in a very abstract way that is not possible with many other languages. The *alpaka* library extensively makes use of advanced functional C++ template meta-programming techniques. The Implementation Details section discusses the C++ library and the way it provides extensibility and optimizability.

5.1 Structure

The *alpaka* library allows offloading of computations from the host execution domain to the accelerator execution domain, whereby they are allowed to be identical.

In the abstraction hierarchy the library code is interleaved with user supplied code as is depicted in the following figure.



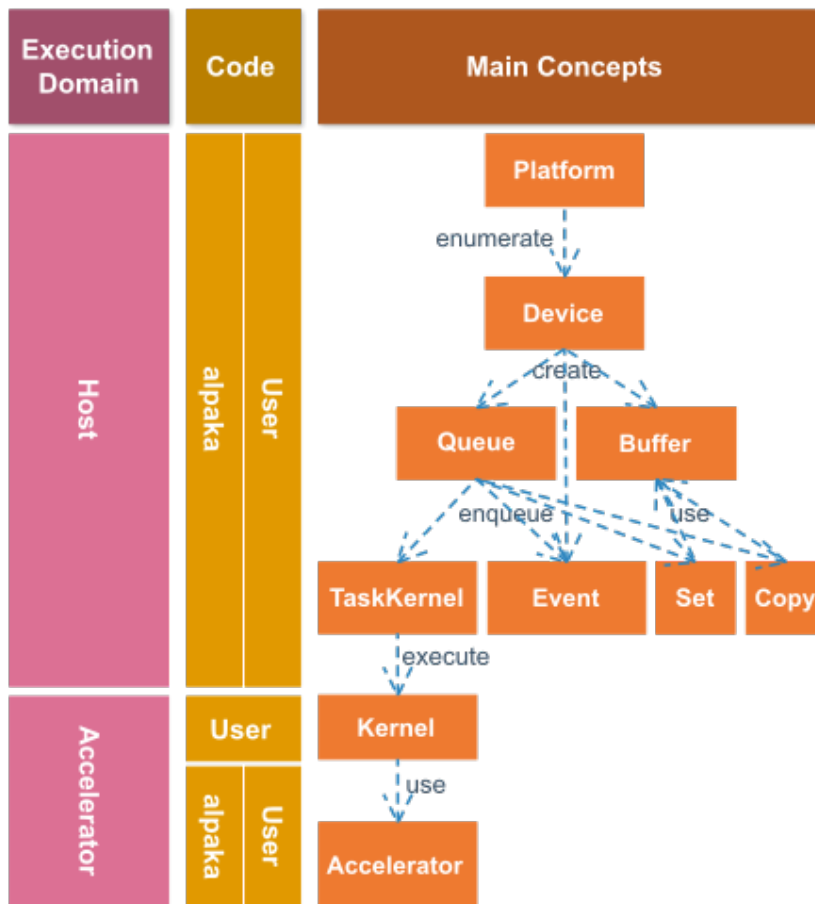
User code invokes library functions, which in turn execute the user provided thread function (kernel) in parallel on the accelerator. The kernel in turn calls library functions when accessing accelerator properties and methods. Additionally, the user can enhance or optimize the library implementations by extending or replacing specific parts.

The *alpaka* abstraction itself only defines requirements a type has to fulfill to be usable with the template functions the library provides. These type constraints are called concepts in C++.

A concept is a set of requirements consisting of valid expressions, associated types, invariants, and complexity guarantees. A type that satisfies the requirements is said to model the concept. A concept can extend the requirements of another concept, which is called refinement. [BoostConcepts](#)

Concepts allow to safely define polymorphic algorithms that work with objects of many different types.

The *alpaka* library implements a stack of concepts and their interactions modeling the abstraction defined in the previous chapter. Furthermore, default implementations for various devices and accelerators modeling those are included in the library. The interaction of the main user facing concepts can be seen in the following figure.



For each type of Device there is a Platform for enumerating the available Device`s. A ``Device is the requirement for creating Queues and Events as it is for allocating Buffers on the respective Device. Buffers can be copied, their memory be set and they can be pinned or mapped. Copying and setting a buffer requires the corresponding Copy and Set tasks to be enqueued into the Queue. An Event can be enqueued into a Queue and its completion state can be queried by the user. It is possible to wait for (synchronize with) a single Event, a Queue or a whole Device. An Executor can be enqueued into a Queue and will execute the Kernel (after all previous tasks in the queue have been completed). The Kernel in turn has access to the Accelerator it is running on. The Accelerator provides the Kernel with its current index in the block or grid, their extents or other data as well as it allows to allocate shared memory, execute atomic operations and many more.

5.2 Interface Usage

5.2.1 Accelerator Functions

Functions that should be executable on an accelerator have to be annotated with the execution domain (one of ALPAKA_FN_HOST, ALPAKA_FN_ACC and ALPAKA_FN_HOST_ACC). They most probably also require access to the accelerator data and methods, such as indices and extents as well as functions to allocate shared memory and to synchronize all threads within a block. Therefore the accelerator has to be passed in as a templated constant reference parameter as can be seen in the following code snippet.

```

template<
    typename TAcc>
ALPAKA_FN_ACC auto doSomethingOnAccelerator(
    TAcc const & acc/*,
    ...*/)                                // Arbitrary number of parameters
-> int                                    // Arbitrary return type
{

```

(continues on next page)

(continued from previous page)

```
//...
}
```

5.2.2 Kernel Definition

A kernel is a special function object which has to conform to the following requirements:

- it has to fulfill the `std::is_trivially_copyable` trait (has to be copyable via memcopy)
- the `operator()` is the kernel entry point * it has to be an accelerator executable function * it has to return `void` * its first argument has to be the accelerator (templated for arbitrary accelerator back-ends) * all other arguments must fulfill `std::is_trivially_copyable`

The following code snippet shows a basic example of a kernel function object.

```
struct MyKernel
{
    template<
        typename TAcc>           // Templated on the accelerator type.
        ALPAKA_FN_ACC           // Macro marking the function to be executable on all
        ↪ accelerators.
        auto operator()(         // The function / kernel to execute.
            TAcc const & acc /*, // The specific accelerator implementation.
            ... */) const        // Must be 'const'.
        -> void
        {
            //...
        }

        // Class can have members but has to be std::is_trivially_
        ↪ copyable.
        // Classes must not have pointers or references to host memory!
};
```

The kernel function object is shared across all threads in all blocks. Due to the block execution order being undefined, there is no safe and consistent way of altering state that is stored inside of the function object. Therefore, the `operator()` of the kernel function object has to be `const` and is not allowed to modify any of the object members.

Kernels can also be defined via lambda expressions.

```
auto kernel = [] ALPAKA_FN_ACC (auto const & acc /* , ... */) -> void {
    // ...
}
```

Attention: NVIDIA's `nvcc` compiler does not support generic lambdas which are marked with `__device__`, which is what `ALPAKA_FN_ACC` expands to (among others) when the CUDA backend is active. Therefore, a workaround is required. The type of the `acc` must be defined outside the lambda.

```
int main() {
    // ...
    using Acc = alpaka::ExampleDefaultAcc<Dim, Idx>;

    auto kernel = [] ALPAKA_FN_ACC (Acc const & acc /* , ... */) -> void {
        // ...
    }
    // ...
}
```

However, the kernel is no longer completely generic and cannot be used with different accelerators. If this is required, the kernel must be defined as a function object.

5.2.3 Index and Work Division

The `alpaka::getWorkDiv` and the `alpaka::getIdx` functions both return a vector of the dimensionality the accelerator has been defined with. They are parametrized by the origin of the calculation as well as the unit in which the values are calculated. For example, `alpaka::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc)` returns a vector with the extents of the grid in units of threads.

5.2.4 Memory fences

Note: Memory fences should not be mistaken for synchronization functions between threads. They solely enforce the ordering of certain memory instructions (see below) and restrict how other threads can observe this order. If you need to rely on the results of memory operations being visible to other threads you must use `alpaka::syncBlockThreads` or atomic functions instead.

The `alpaka::mem_fence` function can be used inside an alpaka kernel to issue a memory fence instruction. This guarantees the following **for the local thread** and regardless of global or shared memory:

- All loads that occur before the fence will happen before all loads occurring after the fence, i.e. no *LoadLoad* reordering.
- All stores that occur before the fence will happen before all stores occurring after the fence, i.e. no *StoreStore* reordering.
- The order of stores will be visible to other threads inside the scope (but not necessarily their results).

Note: `alpaka::mem_fence` does not guarantee that there will be no *LoadStore* reordering. Depending on the back-end, loads occurring before the fence may still be reordered with stores occurring after the fence.

Memory fences can be issued on the block level (`alpaka::memory_scope::Block`), grid level (`alpaka::memory_scope::Grid`) and the device level (`alpaka::memory_scope::Device`). Depending on the memory scope, the *StoreStore* order will be visible to other threads in the same block, in the same grid (*_i.e._* within the same kernel launch), or on the whole device (*_i.e._* across concurrent kernel launches).

Some accelerators (like GPUs) follow weaker cache coherency rules than x86 CPUs. In order to avoid storing to (or loading from) a cache or register it is necessary to prefix all observed buffers with `ALPAKA_DEVICE_VOLATILE_`. This enforces that all loads / stores access the actual global / shared memory location.

Example:

```
/* Initial values:
 * vars[0] = 1
 * vars[1] = 2
 */
template<typename TAcc>
ALPAKA_FN_ACC auto operator()(TAcc const& acc, bool* success, ALPAKA_DEVICE_VOLATILE_
    ↪int* vars) const -> void
{
    auto const idx = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u];

    // Global thread 0 is producer
    if(idx == 0)
    {
        vars[0] = 10;
        alpaka::mem_fence(acc, alpaka::memory_scope::Device{});
        vars[1] = 20;
    }
}
```

(continues on next page)

(continued from previous page)

```

}

auto const b = vars[1];
alpaka::mem_fence(acc, alpaka::memory_scope::Device{});
auto const a = vars[0];

/* Possible results at this point:
 * a == 1 && b == 2
 * a == 10 && b == 2
 * a == 10 && b == 20
 *
 * but NOT:
 * a == 1 && b == 20
 */
}

```

5.2.5 Memory Management

The memory allocation function of the *alpaka* library (`alpaka::allocBuf<TElem>(device, extents)`) is uniform for all devices, even for the host device. It does not return raw pointers but reference counted memory buffer objects that remove the necessity for manual freeing and the possibility of memory leaks. Additionally the memory buffer objects know their extents, their pitches as well as the device they reside on. This allows buffers that possibly reside on different devices with different pitches to be copied only by providing the buffer objects as well as the extents of the region to copy (`alpaka::memcpy(bufDevA, bufDevB, copyExtents)`).

5.2.6 Kernel Execution

The following source code listing shows the execution of a kernel by enqueueing the execution task into a queue.

```

// Define the dimensionality of the task.
using Dim = alpaka::DimInt<1u>;
// Define the type of the indexes.
using Idx = std::size_t;
// Define the accelerator to use.
using Acc = alpaka::AccCpuSerial<Dim, Idx>;
// Select the queue type.
using Queue = alpaka::QueueCpuNonBlocking;

// Select a device to execute on.
auto platformAcc = alpaka::Platform<Acc>{};
auto devAcc = alpaka::getDevByIdx(platformAcc, 0);
// Create a queue to enqueue the execution into.
Queue queue(devAcc);

// Create a 1-dimensional work division with 256 blocks a 16 threads.
auto const workDiv = alpaka::WorkDivMembers<Dim, Idx>(256u, 16u);
// Create an instance of the kernel function object.
MyKernel kernel;
// Enqueue the execution task into the queue.
alpaka::exec<Acc>(queue, workDiv, kernel/*, arguments ...*/);

```

The dimensionality of the task as well as the type for index and extent have to be defined explicitly. Following this, the type of accelerator to execute on, as well as the type of the queue have to be defined. For both of these types instances have to be created. For the accelerator this has to be done indirectly by enumerating the required device via the device manager, whereas the queue can be created directly.

To execute the kernel, an instance of the kernel function object has to be constructed. Following this, an execution task combining the work division (grid and block sizes) with the kernel function object and the bound invocation arguments has to be created. After that this task can be enqueued into a queue for immediate or later execution (depending on the queue used).

CHEATSHEET

6.1 General

- Getting alpaka: <https://github.com/alpaka-group/alpaka>
- Issue tracker, questions, support: <https://github.com/alpaka-group/alpaka/issues>
- All alpaka names are in namespace alpaka and header file *alpaka/alpaka.hpp*
- This document assumes

```
#include <alpaka/alpaka.hpp>
using namespace alpaka;
```

6.2 Accelerator, Platform and Device

Define in-kernel thread indexing type

```
using Dim = DimInt<constant>;
using Idx = IntegerType;
```

Define accelerator type (CUDA, OpenMP, etc.)

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

AcceleratorType:

```
AccGpuCudaRt,
AccGpuHipRt,
AccCpuSycl,
AccFpgaSyclIntel,
AccGpuSyclIntel,
AccCpuOmp2Blocks,
AccCpuOmp2Threads,
AccCpuTbbBlocks,
AccCpuThreads,
AccCpuSerial
```

Create platform and select a device by index

```
auto const platform = Platform<Acc>{};
auto const device = getDevByIdx(platform, index);
```

6.3 Queue and Events

Create a queue for a device

```
using Queue = Queue<Acc, Property>;  
auto queue = Queue{device};
```

Property:

```
Blocking  
NonBlocking
```

Put a task for execution

```
enqueue(queue, task);
```

Wait for all operations in the queue

```
wait(queue);
```

Create an event

```
Event<Queue> event{device};
```

Put an event to the queue

```
enqueue(queue, event);
```

Check if the event is completed

```
isComplete(event);
```

Wait for the event (and all operations put to the same queue before it)

```
wait(event);
```

6.4 Memory

Memory allocation and transfers are symmetric for host and devices, both done via alpaka API

Create a CPU device for memory allocation on the host side

```
auto const platformHost = PlatformCpu{};  
auto const devHost = getDevByIdx(platformHost, 0);
```

Allocate a buffer in host memory

```
Vec<Dim, Idx> extent = value;  
using BufHost = Buf<DevHost, DataType, Dim, Idx>;  
BufHost bufHost = allocBuf<DataType, Idx>(devHost, extent);
```

(Optional, affects CPU – GPU memory copies) Prepare it for asynchronous memory copies

```
prepareForAsyncCopy(bufHost);
```

Create a view to host memory represented by a pointer

```
using Dim = alpaka::DimInt<1u>;
Vec<Dim, Idx> extent = size;
DataType* ptr = ...;
auto hostView = createView(devHost, ptr, extent);
```

Create a view to host `std::vector`

```
auto vec = std::vector<DataType>(42u);
auto hostView = createView(devHost, vec);
```

Create a view to host `std::array`

```
std::array<DataType, 2> array = {42u, 23};
auto hostView = createView(devHost, array);
```

Get a raw pointer to a buffer or view initialization, etc.

```
DataType* raw = view::getPtrNative(bufHost);
DataType* rawViewPtr = view::getPtrNative(hostView);
```

Allocate a buffer in device memory

```
auto bufDevice = allocBuf<DataType, Idx>(device, extent);
```

Enqueue a memory copy from host to device

```
memcpy(queue, bufDevice, bufHost, extent);
```

Enqueue a memory copy from device to host

```
memcpy(queue, bufHost, bufDevice, extent);
```

6.5 Kernel Execution

Automatically select a valid kernel launch configuration

```
Vec<Dim, Idx> const globalThreadExtent = vectorValue;
Vec<Dim, Idx> const elementsPerThread = vectorValue;

auto autoWorkDiv = getValidWorkDiv<Acc>(
    device,
    globalThreadExtent, elementsPerThread,
    false,
    GridBlockExtentSubDivRestrictions::Unrestricted);
```

Manually set a kernel launch configuration

```
Vec<Dim, Idx> const blocksPerGrid = vectorValue;
Vec<Dim, Idx> const threadsPerBlock = vectorValue;
Vec<Dim, Idx> const elementsPerThread = vectorValue;

using WorkDiv = WorkDivMembers<Dim, Idx>;
auto manualWorkDiv = WorkDiv{blocksPerGrid,
    threadsPerBlock,
    elementsPerThread};
```

Instantiate a kernel and create a task that will run it (does not launch it yet)

```
Kernel kernel{argumentsForConstructor};
auto taskRunKernel = createTaskKernel<Acc>(workDiv, kernel, parameters);
```

acc parameter of the kernel is provided automatically, does not need to be specified here

Put the kernel for execution

```
enqueue(queue, taskRunKernel);
```

6.6 Kernel Implementation

Define a kernel as a C++ functor

```
struct Kernel {
    template<typename Acc>
        ALPAKA_FN_ACC void operator()(Acc const & acc, parameters) const { ... }
};
```

ALPAKA_FN_ACC is required for kernels and functions called inside, acc is mandatory first parameter, its type is the template parameter

Access multi-dimensional indices and extents of blocks, threads, and elements

```
auto idx = getId<Origin, Unit>(acc);
auto extent = getWorkDiv<Origin, Unit>(acc);
// Origin: Grid, Block, Thread
// Unit: Blocks, Threads, Elms
```

Access components of and destructure multi-dimensional indices and extents

```
auto idxX = idx[0];
auto [z, y, x] = extent3D;
```

Linearize multi-dimensional vectors

```
auto linearIdx = mapIdx<lu>(idx, extent);
```

Allocate static shared memory variable

```
Type& var = declareSharedVar<Type, __COUNTER__>(acc); // scalar
auto& arr = declareSharedVar<float[256], __COUNTER__>(acc); // array
```

Get dynamic shared memory pool, requires the kernel to specialize

```
trait::BlockSharedMemDynSizeBytes
    Type * dynamicSharedMemoryPool = getDynSharedMem<Type>(acc);
```

Synchronize threads of the same block

```
syncBlockThreads(acc);
```

Atomic operations

```
auto result = atomicOp<Operation>(acc, arguments);
// Operation: AtomicAdd, AtomicSub, AtomicMin, AtomicMax, AtomicExch,
//             AtomicInc, AtomicDec, AtomicAnd, AtomicOr, AtomicXor, AtomicCas
// Also dedicated functions available, e.g.:
auto old = atomicAdd(acc, ptr, 1);
```

Memory fences on block-, grid- or device level (guarantees LoadLoad and StoreStore ordering)

```
mem_fence(acc, memory_scope::Block{});  
mem_fence(acc, memory_scope::Grid{});  
mem_fence(acc, memory_scope::Device{});
```

Warp-level operations

```
uint64_t result = warp::ballot(acc, idx == 1 || idx == 4);  
assert( result == (1<<1) + (1<<4) );  
  
int32_t valFromSrcLane = warp::shfl(val, srcLane);
```

Math functions take acc as additional first argument

```
math::sin(acc, argument);
```

Similar for other math functions.

Generate random numbers

```
auto distribution = rand::distribution::createNormalReal<double>(acc);  
auto generator = rand::engine::createDefault(acc, seed, subsequence);  
auto number = distribution(generator);
```


RATIONALE

7.1 Interface Distinction

The *alpaka* library is different from other similar libraries (especially *CUDA*) in that it refrains from using implicit or hidden state. This and other interface design decisions will be explained in the following paragraphs.

7.1.1 No Current Device:

The *CUDA* runtime API for example supplies a current device for each user code kernel-thread. Working with multiple devices requires to call `cudaSetDevice` to change the current device whenever an operation should be executed on a non-current device. Even the functions for creating a queue (`cudaStreamCreate`) or an event (`cudaEventCreate`) use the current device without any way to create them on a non current device. In the case of an event this dependency is not obvious, since at the same time queues can wait for events from multiple devices allowing cross-device synchronization without any additional work. So conceptually an event could also have been implemented device independently. This can lead to hard to track down bugs due to the non-explicit dependencies, especially in multi-threaded code using multiple devices.

7.1.2 No Default Device:

In contrast to the *CUDA* runtime API *alpaka* does not provide a device by default per kernel-thread. Especially in combination with *OpenMP* parallelized host code this keeps users from surprises. The following code snippet shows that it does not necessarily do what one would expect.

```
1  cudaSetDevice(1);
2
3  #pragma omp parallel for
4  for(int i = 0; i<10; ++i)
5  {
6      kernel<<<blocks,threads>>>(i);
7  }
```

Depending on what the *CUDA* runtime API selects as default device for each of the *OpenMP* threads (due to each of them having its own current device), not all of the kernels will necessarily run on device one.

In the *alpaka* library all such dependencies are made explicit. All functions depending on a device require it to be given as a parameter. The *alpaka CUDA* back-end checks before forwarding the calls to the *CUDA* runtime API whether the current device matches the given one and changes it if required. The *alpaka CUDA* back-end does not reset the current device to the one prior to the method invocation out of performance considerations. This has to be considered when native *CUDA* code is combined with *alpaka* code.

7.1.3 No Default Queue:

CUDA allows to execute commands without specifying a queue. The default queue that is used synchronizes implicitly with all other queues on the device. If a command queue is issued to the default, all other asynchronous queues have to wait before executing any new commands, even when they have been enqueued much earlier. This can introduce hard to track down performance issues. As of *CUDA* 7.0 the default queue can be converted to a non synchronizing queue with a compiler option. Because concurrency is crucial for performance and users should think about the dependencies between their commands from begin on, *alpaka* does not provide such a default queue. All asynchronous operations (kernel launches, memory copies and memory sets) require a queue to be executed in.

7.2 No Implicit Built-in Variables and Functions:

Within *CUDA* device functions (functions annotated with `__global__` or `__device__`) built-in functions (`__sync_threads`, `__threadfence`, `atomicAdd`, ...) and variables (`gridDim`, `blockIdx`, `blockDim`, `threadIdx`, `warpSize`, ...) are provided.

It would have been possible to emulate those implicit definitions by forcing the kernel function object to inherit from a class providing these functions and members. However functions outside the kernel function object would then pose a problem. They do not have access to those functions and members, the function object has inherited. To circumvent this, the functions and members would have to be public, the inheritance would have to be public and a reference to the currently executing function object would have to be passed as parameter to external functions. This would have been too cumbersome and inconsistent. Therefore access to the accelerator is given to the user kernel function object via one special input parameter representing the accelerator. After that this accelerator object can simply be passed to other functions. The built-in variables can be accessed by the user via query functions on this accelerator.

- Abandoning all the implicit and default state makes it much easier for users of the library to reason about their code. *

7.3 No Language Extensions:

Unlike *CUDA*, the *alpaka* library does not extend the C++ language with any additional variable qualifiers (`__shared__`, `__constant__`, `__device__`) defining the memory space. Instead of those qualifiers *alpaka* provides accelerator functions to allocate memory in different the different memory spaces.

7.4 No Dimensionality Restriction:

CUDA always uses three-dimensional indices and extents, even though the task may only be one or two dimensional. *OpenCL* on the other hand allows grid and block dimensions in the range [1,3] but does not provide corresponding n-dimensional indices, but rather provides functions like `get_global_id` or `get_local_id`, which require the dimension in which the one-dimensional ID is to be queried as a parameter. By itself this is no problem, but how can be assured that a two-dimensional kernel is called with grid and block extents of the correct dimensionality at compile time? How can it be assured that a kernel which only uses `threadIdx.x` or equivalently calls `get_global_id(0)` will not get called with two dimensional grid and block extents? Because the result in such a case is undefined, and most of the time not wanted by the kernel author, this should be easy to check and reject at compile-time. In *alpaka* all accelerators are templated on the dimensionality. This allows a two-dimensional image filter to assert that it is only called with a two dimensional accelerator. Thereby the algorithms can check for supported dimensionality of the accelerator at compile time instead of runtime. Furthermore with the dimension being a template parameter, the CPU back-end implementations are able to use only the number of nested loops really necessary instead of the 6 loops (2 x 3 loops for grid blocks and block threads), which are mandatory to emulate the *CUDA* threaded blocking scheme.

By hiding all the accelerator functionality inside of the accelerator object that is passed to the user kernel, the user of the **alpaka* library is not faced with any non-standard C++ extensions. Nevertheless the *CUDA* back-end internally uses those language extensions.*

7.5 Integral Sizes of Arbitrary Type:

The type of sizes such as extents, indices and related variables are depending on a template parameter of the accelerator and connected classes. This allows the kernel to be executed with sizes of arbitrary ranges. Thereby it is possible to force the accelerator back-ends to perform all internal index, extent and other integral size depending computations with a given precision. This is especially useful on current *NVIDIA* GPUs. Even though they support 64-bit integral operations, they are emulated with multiple 32-bit operations. This can be a huge performance penalty when the sizes of buffers, offsets, indices and other integral variables holding sizes are known to be limited.

7.6 No Synchronous (Blocking) and Asynchronous (Non-Blocking) Function Versions:

CUDA provides two versions of many of the runtime functions, for example, *cudaMemcpyAsync* and *cudaMemcpy*. The asynchronous version requires a queue while the synchronous version does not need a queue parameter. The asynchronous version immediately returns control back to the caller while the task is enqueued into the given queue and executed later in parallel to the host code. The synchronous version waits for the task to finish before the function call returns control to the caller. Inconsistently, all kernels in a *CUDA* program can only be started either asynchronously by default or synchronously if *CUDA_LAUNCH_BLOCKING* is defined. There is no way to specify this on a per kernel basis. To switch a whole application from asynchronous to synchronous calls, for example for debugging reasons, it is necessary to change the names of all the runtime functions being called as well as their parameters. In *alpaka* this is solved by always enqueueing all tasks into a queue and not defining a default queue. 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. There is no need to change any line of calling code.

7.7 Memory Management

Memory buffers can not only be identified by the pointer to their first byte. The C++ *new* and *malloc*, the *CUDA* *cudaMalloc* as well as the *OpenCL* *clCreateBuffer* functions all return a plain pointer. This is not enough when working with multiple accelerators and multiple devices. To know where a specific pointer was allocated, additional information has to be stored to uniquely identify a memory buffer on a specific device. Memory copies between multiple buffers additionally require the buffer extents and pitches to be known. Many APIs, for example *CUDA*, require the user to store this information externally. To unify the usage, *alpaka* stores all the necessary information in a memory buffer object.

7.8 Acceleratable Functions

Many parallelization libraries / frameworks do not fully support the separation of the parallelization strategy from the algorithm itself. *OpenMP*, for example, fully mixes the per thread algorithm and the parallelization strategy. This can be seen in the source listing showing a simple AXPY computation with *OpenMP*.

```

1 template<
2     typename TIdx,
3     typename TElem>
4 void axpyOpenMP(
5     TIdx const n,
```

(continues on next page)

(continued from previous page)

```

6      TElem const alpha,
7      TElem const * const X,
8      TElem * const Y)
9  {
10     #pragma omp parallel for
11     for (i=0; i<n; i++)
12     {
13         Y[i] = alpha * X[i] + Y[i];
14     }
15 }

```

Only one line of the function body, line 13, is the algorithm itself, while all surrounding lines represent the parallelization strategy.

CUDA, *OpenCL* and other libraries allow, at least to some degree, to separate the algorithm from the parallelization strategy. They define the concept of a kernel representing the algorithm itself which is then parallelized depending on the underlying hardware. The AXPY *CUDA* kernel source code shown in figure consists only of the code of one single iteration.

```

1  template<
2      typename TIdx,
3      typename TElem>
4  __global__ void axpyCUDA(
5      TIdx const n,
6      TElem const alpha,
7      TElem const * const X,
8      TElem * const Y)
9  {
10     TIdx const i(blockIdx.x*blockDim.x + threadIdx.x)
11     if(i < n)
12     {
13         Y[i] = alpha * X[i] + Y[i];
14     }
15 }

```

On the other hand the *CUDA* implementation is bloated with code handling the inherent blocking scheme. Even if the algorithm does not utilize blocking, as it is the case here, the algorithm writer has to calculate the global index of the current thread by hand (line 10). Furthermore, to support vectors larger than the predefined maximum number of threads per block (1024 for current *CUDA* devices), multiple blocks have to be used. When the number of blocks does not divide the number of vector elements, it has to be assured that the threads responsible for the vector elements behind the given length, do not access the memory to prevent a possible memory access error.

By using the kernel concept, the parallelization strategy, whether all elements are executed in sequential order, in parallel or blocked is not hard coded into the algorithm itself. The possibly multidimensional nested loops do not have to be written by the user. For example, six loops would be required to emulate the *CUDA* execution pattern with a grid of blocks consisting of threads.

Furthermore the kernel concept breaks the algorithm down to the per element level. Recombining multiple kernel iterations to loop over lines, columns, blocks or any other structure is always possible by changing the calling code and does not require a change of the kernel. In contrast, by using *OpenMP* this would not be possible. Therefore the *alpaka* interface builds on the kernel concept, being the body of the corresponding standard for loop executed in each thread.

7.9 Execution Domain Specifications

CUDA requires the user to annotate its functions with execution domain specifications. Functions that can only be executed on the GPU have to be annotated with `__device__`, functions that can be executed on the host and on the GPU have to be annotated with `__host__ __device__` and host only functions can optionally be annotated with `__host__`. The *nvcc* *CUDA* compiler uses these annotations to decide with which back-ends a function has to be compiled. Depending on the compiler in use, *alpaka* defines the macros `ALPAKA_FN_HOST`, `ALPAKA_FN_ACC` and `ALPAKA_FN_HOST_ACC` with the identical meaning which can be used in the same positions. When the *CUDA* compiler is used, they are defined to their *CUDA* equivalents, else they are empty.

7.10 Kernel Function

7.10.1 Requirements

- User kernels should be implemented independent of the accelerator.
- A user kernel has to have access to accelerator methods (synchronization within blocks, index retrieval, ...).
- For usage with *CUDA*, the kernel methods have to be attributed with `__device__ __host__`.
- The user kernel has to fulfill `std::is_trivially_copyable` because only such objects can be copied into *CUDA* device memory. A trivially copyable class is a class that #. Has no non-trivial copy constructors (this also requires no virtual functions or virtual bases) #. Has no non-trivial move constructors #. Has no non-trivial copy assignment operators #. Has no non-trivial move assignment operators #. Has a trivial destructor
- For the same reason all kernel parameters have to fulfill `std::is_trivially_copyable`, too.

7.10.2 Implementation Variants

There are two possible ways to tell the kernel about the accelerator type:

1. The kernel is templated on the accelerator type ...
 - (+) This allows users to specialize them for different accelerators. (Is this is really necessary or desired?)
 - (-) The kernel has to be a class template. This does not allow C++ lambdas to be used as kernels because they are no templates themselves (but only their `operator()` can be templated).
 - (-) This prevents the user from instantiating an accelerator independent kernel before executing it. Because the memory layout in inheritance hierarchies is undefined a simple copy of the user kernel or its members to its specialized type is not possible platform independently. This would require a copy from `UserKernel<TDummyAcc>` to `UserKernel<TAcc>` to be possible. The only way to allow this would be to require the user to implement a templated copy constructor for every kernel. This is not allowed for kernels that should be copyable to a *CUDA* device because `std::is_trivially_copyable` requires the kernel to have no non-trivial copy constructors.
 - a) ... and inherits from the accelerator.
 - (-) The kernel itself has to inherit at least protected from the accelerator to allow the `KernelExecutor` to access the `Accelerator`.
 - (-) How do accelerator functions called from the kernel (and not within the kernel class itself) access the accelerator methods?

Casting this to the accelerator type and giving it as parameter is too much to require from the user.
 - b) ... and the `operator()` has a reference to the accelerator as parameter.
 - (+) This allows to use the accelerator in functions called from the kernel (and not within the kernel class itself) to access the accelerator methods in the same way the kernel entry point function can.

- (-) This would require an additional object (the accelerator) in device memory taking up valuable CUDA registers (opposed to the inheritance solution). At least on CUDA all the accelerator functions could be inlined nevertheless.
2. The `operator()` is templated on the accelerator type and has a reference to the accelerator as parameter.
- (+) The kernel can be an arbitrary function object with `ALPAKA_FN_HOST_ACC` attributes.
 - (+) This would allow to instantiate the accelerator independent kernel and set its members before execution.
 - (+/-) usable with polymorphic lambdas.
 - (-) The `operator()` could be overloaded on the accelerator type but there is no way to specialize the whole kernel class itself, so it always has the same members.
 - (-) This would require an additional object (the accelerator) in device memory taking up valuable CUDA registers (opposed to the inheritance solution). At least on CUDA all the accelerator functions could be inlined nevertheless.

Currently we implement version 2.

7.10.3 Implementation Notes

Unlike *CUDA*, the *alpaka* library does not differentiate between the kernel function that represents the entry point and other functions that can be executed on the accelerator. The entry point function that has to be annotated with `__global__` in *CUDA* is internal to the *alpaka CUDA* back-end and is not exposed to the user. It directly calls into the user supplied kernel function object whose invocation operator is declared with `ALPAKA_FN_ACC`, which equals `__device__` in *CUDA*. In this respect there is no difference between the kernel entry point function and any other accelerator function in *alpaka*.

The `operator()` of the kernel function object has to be `const`. This is especially important for the *CUDA* back-end, as it could possibly use the constant memory of the GPU to store the function object. The constant memory is a fast, cached, read-only memory that is beneficial when all threads uniformly read from the same address at the same time. In this case it is as fast as a read from a register.

7.10.4 Access to Accelerator-Dependent Functionality

There are two possible ways to implement access to accelerator dependent functionality inside a kernel:

- Making the functions/templates members of the accelerator (maybe by inheritance) and calling them like `acc.syncBlockThreads()` or `acc.template getIdIdx<Grid, Thread, Dim1>()`. This would require the user to know and understand when to use the template keyword inside dependent type object function calls.
- The functions are only light wrappers around traits that can be specialized taking the accelerator as first value (it can not be the last value because of the potential use of variadic arguments). The resulting code would look like `sync(acc)` or `getIdIdx<Grid, Thread, Dim1>(acc)`. Internally these wrappers would call trait templates that are specialized for the specific accelerator e.g. `template<typename TAcc> Sync{...};`

The second version is easier to understand and usually shorter to use in user code.

7.11 Index and Work Division

CUDA requires the user to calculate the global index of the current thread within the grid by hand (already shown as `axpyCUDA`). On the contrary, *OpenCL* provides the methods `get_global_size`, `get_global_id`, `get_local_size` and `get_local_id`. Called with the required dimension, they return the corresponding local or global index or extent (size). In *alpaka* this idea is extended to all dimensions. To unify the method interface and to avoid confusion between the differing terms and meanings of the functions in *OpenCL* and *CUDA*, in *alpaka* these methods are template functions.

7.12 Block Shared Memory

7.12.1 Static Block Shared Memory

The size of block shared memory that is allocated inside the kernel is required to be given as compile time constant. This is due to *CUDA* not allowing to allocate block shared memory inside a kernel at runtime.

7.12.2 Dynamic Block Shared Memory

The size of the external block shared memory is obtained from a trait that can be specialized for each kernel. The trait is called with the current kernel invocation parameters and the block-element extent prior to each kernel execution. Because the block shared memory size is only ever constant or dependent on the block-element extent or the parameters of the invocation this has multiple advantages:

- It forces the separation of the kernel invocation from the calculation of the required block shared memory size.
- It lets the user write this calculation once instead of multiple times spread across the code.

MAPPING ONTO SPECIFIC HARDWARE ARCHITECTURES

By providing an accelerator independent interface for kernels, their execution and memory accesses at different hierarchy levels, *alpaka* allows the user to write accelerator independent code that does not neglect performance.

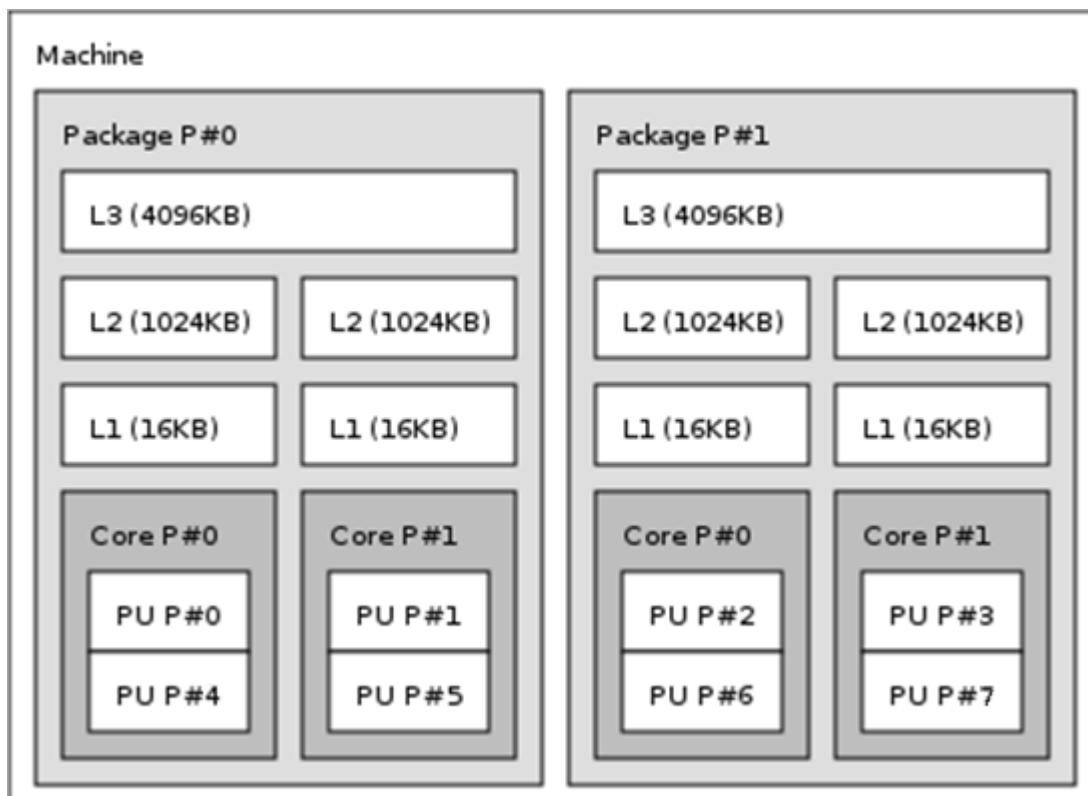
The mapping of the decomposition to the execution environment is handled by the back-ends provided by the *alpaka* library as well as user defined back-ends. A computation that is described with a maximum of the parallelism available in the *redundant hierarchical parallelism* abstraction can not be mapped one to one to any existing hardware. GPUs do not have vector registers for `float` or `double` types. Therefore, the element level is often omitted on *CUDA* accelerators. CPUs in turn are not (currently) capable of running thousands of threads concurrently and do not have equivalently fast inter-thread synchronization and shared memory access as GPUs do.

A major point of the *redundant hierarchical parallelism* abstraction is to ignore specific unsupported levels and utilize only the ones supported on a specific accelerator. This allows a mapping to various current and future accelerators in a variety of ways enabling optimal usage of the underlying compute and memory capabilities.

The grid level is always mapped to the whole device being in consideration. The scheduler can always execute multiple kernel grids from multiple queues in parallel by statically or dynamically subdividing the available resources. However, this will only ever simplify the mapping due to less available processing units. Furthermore, being restricted to less resources automatically improves the locality of data due to spatial and temporal locality properties of the caching hierarchy.

8.1 x86 CPUs

There are multiple possible ways to map the *alpaka* abstraction to x86 CPUs. The following figure shows the compute and memory hierarchy of a dual-socket (package) node with dual-core CPUs and symmetric multithreading (Hyper-Threading). Through symmetric multithreading (Hyper-Threading) each core represents two processing units.



8.1.1 Thread

Mapping the thread level directly to the processing units is the most trivial part of the assignment of hierarchy levels to hardware units. However, the block and warp levels could be mapped to hardware components in different ways with varying advantages and disadvantages.

8.1.2 Warp

Even though a warp seems to be identical to a vector register, because both execute a single uniform instruction on multiple data elements, they are not the same. *Warps* can handle branches with divergent control flows of multiple threads. There is no equivalent hardware unit in a CPU supporting this. Therefore, the warp level can not be utilized on CPUs leading to a one-to-one mapping of threads to warps which does not violate the rules of the abstraction.

8.1.3 Block

One Block Per Node

By combining all processing units (possibly Hyper-Threads) of all processors on a node into one block, the number of synchronizing and communicating threads can be enlarged. This high possible thread count would simplify the implementation of some types of algorithms but introduces performance issues on multi-core nodes. The shared memory between all cores on a node is the RAM. However, the RAM and the communication between the sockets is far too slow for fine-grained communication in the style of *CUDA* threads.

One Block Per Socket

If each processor on each socket would concurrently execute one block, the L3 cache would be used as the fast shared memory. Although this is much better than to use the RAM, there is still a problem. Regions of the global memory and especially from the shared memory that are accessed are automatically cached in the L1 and / or L2 caches of each core. Not only the elements which are directly accessed will be cached but always the whole cache line they lie in. Cache lines typically have a size of 64 Bytes on modern x86 architectures. This leads to, for example, eight double precision floating point numbers being cached at once even though only one value really is required. As long as these values are only read there is no problem. However, if one thread writes to a value that is also cached on other cores, all such cache copies have to be invalidated. This results in a lot of cache and bus traffic. Due to the hierarchical decomposition of the grid of threads reflecting the data elements, neighboring threads are always combined into a common block. By mapping a block to a socket, threads that are executed concurrently always have very close indices into the grid. Therefore, the elements that are read and written by the threads are always very close together within the memory and will most probably share a cache line. This property is exploited on *CUDA* GPUs, where memory accesses within a warp are combined into one large transaction. However, when multiple threads from multiple CPU cores write to different elements within a cache line, this advantage is reversed into its opposite. This pattern non-intuitively leads to heavy performance degradation and is called false-sharing.

One Block Per Core

The best compromise between a high number of threads per block and a fast communication between the threads is to map a block directly to a CPU core. Each processing unit (possibly a Hyper-Thread) executes one or more threads of our hierarchical abstraction while executing multiple elements locally either by processing them sequentially or in a vectorized fashion. This possible mapping of blocks, threads and elements to the compute and memory hierarchy of a dual-socket node with dual-core CPUs and symmetric multithreading is illustrated in the following figure. `![x86_cpu](x86/x86_cpu_mapping.png)`

One Block Per Thread

If there is no symmetric multithreading or if it is desired, it is also possible to implement a mapping of one block with exactly one thread for each processing unit. This allows to completely remove the synchronization overhead for tasks where this is not required at all.

8.1.4 Threading Mechanisms

The mapping of threads to processing units is independent of the threading mechanism that is used. As long as the thread affinity to cores can be set correctly, *OpenMP*, *pthread*, *std::thread* or other libraries and APIs can be used interchangeably to implement various *alpaka* back-ends. They all have different advantages and disadvantages. Real operating system threads like *pthread*, *std::thread* and others have a high cost of thread creation and thread change because their default stack size amounts to multiple megabytes. *OpenMP* threads on the other hand are by default much more lightweight. However, they are arbitrarily limited by the runtime implementation in the maximum number of concurrent threads a machine supports. All of the previous methods have non-deterministic thread changes in common. Therefore it is not possible to decide the order in which threads within a block are processed, which could be a good optimization opportunity.

To allow blocks to contain more threads than the number of processing units each core provides, it is possible to simply start more threads than processing units are available. This is called oversubscription. Those threads can be bound to the correct cores and by relying on the operating system thread scheduler, they are preemptively multitasked while sharing a single cache and thereby avoiding false-sharing. However, this is not always beneficial because the cost of thread changes by the kernel-mode scheduler should not be underestimated.

8.2 GPUs (CUDA/HIP)

Mapping the abstraction to GPUs supporting *CUDA* and *HIP* is straightforward because the hierarchy levels are identical up to the element level. So blocks of warps of threads will be mapped directly to their *CUDA/HIP* equivalent.

The element level is supported through an additional run-time variable containing the extent of elements per thread. This variable can be accessed by all threads and should optimally be placed in constant device memory for fast access.

CMAKE ARGUMENTS

Alpaka configures a large part of its functionality at compile time. Therefore, a lot of compiler and link flags are needed, which are set by CMake arguments. First, we show a simple way to build alpaka for different back-ends using [CMake Presets](#). The second part of the documentation shows the general and back-end specific alpaka CMake flags.

Hint: To display the cmake variables with value and type in the build folder of your project, use `cmake -LH <path-to-build>`.

9.1 CMake Presets

The [CMake Presets](#) are defined in the `CMakePresets.json` file. Each preset contains a set of CMake arguments. We use different presets to build the examples and tests with different back-ends. Execute the following command to display all presets:

```
cd <alpaka_project_root>
cmake --list-presets
```

To configure, build and run the tests of a specific preset, run the following commands (for the example, we use the `cpu-serial` preset):

```
cd <alpaka_project_root>
# configure a specific preset
cmake --preset cpu-serial
# build the preset
cmake --build --preset cpu-serial
# run test of the preset
ctest --preset cpu-serial
```

All presets are configured and built in a subfolder of the `<alpaka_project_root>/build` folder.

9.1.1 Modifying and Extending Presets

The easiest way to change a preset is to set CMake arguments during configuration:

```
cd <alpaka_project_root>
# configure the cpu-serial preset with clang++ as C++ compiler
cmake --preset cpu-serial -DCMAKE_CXX_COMPILER=clang++
# build the preset
cmake --build --preset cpu-serial
# run test of the preset
ctest --preset cpu-serial
```

It is also possible to configure the default setting first and then change the arguments with `ccmake`:

```
cd <alpaka_project_root>
# configure the cpu-serial preset with clang++ as C++ compiler
cmake --preset cpu-serial
cd build/cpu-serial
ccmake .
cd ../../
# build the preset
cmake --build --preset cpu-serial
# run test of the preset
ctest --preset cpu-serial
```

CMake presets also offer the option of creating personal, user-specific configurations based on the predefined CMake presets. To do this, you can create the file `CMakeUserPresets.json` in the root directory of your project (the file is located directly next to `CMakePresets.json`). You can then create your own configurations from the existing CMake presets. The following example takes the `cpu-serial` configuration, uses `ninja` as the generator instead of the standard generator and uses the build type `RELEASE`.

```
{
  "version": 3,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 22,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "cpu-serial-ninja-release",
      "inherits": "cpu-serial",
      "generator": "Ninja",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": {
          "type": "STRING",
          "value": "RELEASE"
        }
      }
    }
  ]
}
```

Hint: Many IDEs like [Visual Studio Code](#) and [CLion](#) support CMake presets.

9.2 Arguments

Table of back-ends

- *CPU Serial*
- *C++ Threads*
- *Intel TBB*
- *OpenMP 2 Grid Block*
- *OpenMP 2 Block Thread*
- *CUDA*

- *HIP*

9.2.1 Common

alpaka_CXX_STANDARD

Set the C++ standard version.

alpaka_BUILD_EXAMPLES

Build the examples.

BUILD_TESTING

Build the testing tree.

alpaka_INSTALL_TEST_HEADER

Install headers of the namespace `alpaka::test`.
Attention, headers are **not** designed **for** production code.
They should only be used **for** prototyping **or** creating tests that use alpaka functionality.

alpaka_DEBUG

Set Debug level:

0 - Is the default value. No additional logging.
1 - Enables some basic flow traces.
2 - Display **as** many information **as** possible. Especially pointers, sizes **and** other parameters of copies, kernel invocations **and** other operations will be **↪** printed.

alpaka_USE_INTERNAL_CATCH2

Use internally shipped Catch2.

alpaka_FAST_MATH

Enable fast-math **in** kernels.

Warning: The default value is changed to “OFF” with alpaka 0.7.0.

alpaka_FTZ

Set flush to zero **for** GPU.

alpaka_DEBUG_OFFLOAD_ASSUME_HOST

Allow host-only constructs like **assert in** offload code **in** debug mode.

alpaka_USE_MDSPAN

Enable/Disable the use of ``std::experimental::mdspan``:

"OFF" - Disable mdspan

(continues on next page)

(continued from previous page)

"SYSTEM" - Enable mdspan and acquire it via ``find_package`` from your system
"FETCH" - Enable mdspan and download it via CMake's ``FetchContent`` from GitHub. 
→ The dependency will not be installed when you install alpaka.

9.2.2 CPU Serial

alpaka_ACC_CPU_B_SEQ_T_SEQ_ENABLE

Enable the serial CPU back-end.

alpaka_BLOCK_SHARED_DYN_MEMBER_ALLOC_KIB

Kibibytes (1024B) of memory to allocate **for** block shared memory **for** backends requiring static allocation.

9.2.3 C++ Threads

alpaka_ACC_CPU_B_SEQ_T_THREADS_ENABLE

Enable the threads CPU block thread back-end.

9.2.4 Intel TBB

alpaka_ACC_CPU_B_TBB_T_SEQ_ENABLE

Enable the TBB CPU grid block back-end.

alpaka_BLOCK_SHARED_DYN_MEMBER_ALLOC_KIB

Kibibytes (1024B) of memory to allocate **for** block shared memory **for** backends requiring static allocation.

9.2.5 OpenMP 2 Grid Block

alpaka_ACC_CPU_B_OMP2_T_SEQ_ENABLE

Enable the OpenMP 2.0 CPU grid block back-end.

alpaka_BLOCK_SHARED_DYN_MEMBER_ALLOC_KIB

Kibibytes (1024B) of memory to allocate **for** block shared memory **for** backends requiring static allocation.

9.2.6 OpenMP 2 Block thread

alpaka_ACC_CPU_B_SEQ_T_OMP2_ENABLE

Enable the OpenMP 2.0 CPU block thread back-end.

9.2.7 CUDA

alpaka_ACC_GPU_CUDA_ENABLE

Enable the CUDA GPU back-end.

alpaka_ACC_GPU_CUDA_ONLY_MODE

Only back-ends using CUDA can be enabled **in** this mode (This allows to mix alpaka code **with** native CUDA code).

CMAKE_CUDA_ARCHITECTURES

Set the GPU architecture: e.g. "35;72".

CMAKE_CUDA_COMPILER

Set the CUDA compiler: "nvcc" (default) **or** "clang++".

CUDACXX

Select a specific CUDA compiler version.

alpaka_CUDA_KEEP_FILES

Keep **all** intermediate files that are generated during internal compilation steps 'CMakeFiles/<targetname>.dir'.

alpaka_CUDA_EXPT_EXTENDED_LAMBDA

Enable experimental, extended host-device lambdas **in** NVCC.

alpaka_RELOCATABLE_DEVICE_CODE

Enable relocatable device code. Note: This affects all targets in the CMake scope where ``alpaka\_RELOCATABLE\_DEVICE\_CODE`` is set. For the effects on CUDA code see NVIDIA's blog post:

<https://developer.nvidia.com/blog/separate-compilation-linking-cuda-device-code/>

alpaka_CUDA_SHOW_CODELINES

Show kernel lines **in** cuda-gdb **and** cuda-memcheck. If alpaka_CUDA_KEEP_FILES **is** enabled source code will be inlined **in** ptx. One of the added flags **is**: --generate-line-info

alpaka_CUDA_SHOW_REGISTER

Show the number of used kernel registers during compilation **and** create PTX.

9.2.8 HIP

To enable the HIP back-end please extend `CMAKE_PREFIX_PATH` with the path to the HIP installation.

`alpaka_ACC_GPU_HIP_ENABLE`

Enable the HIP back-end (**a**ll other back-ends must be disabled).

`alpaka_ACC_GPU_HIP_ONLY_MODE`

Only back-ends using HIP can be enabled **i**n this mode.

`CMAKE_HIP_ARCHITECTURES`

Set the GPU architecture: e.g. `"gfx900;gfx906;gfx908"`.

A list of the GPU architectures can be found [here](#).

`alpaka_HIP_KEEP_FILES`

Keep **a**ll intermediate files that are generated during internal compilation steps `'CMakeFiles/<targetname>.dir'`.

`alpaka_RELOCATABLE_DEVICE_CODE`

Enable relocatable device code. Note: This affects all targets in the CMake scope where ```alpaka_RELOCATABLE_DEVICE_CODE``` is set. For the effects on HIP code see the NVIDIA blog post linked below; HIP follows CUDA's behaviour.

<https://developer.nvidia.com/blog/separate-compilation-linking-cuda-device-code/>

9.2.9 SYCL

`alpaka_RELOCATABLE_DEVICE_CODE`

Enable relocatable device code. Note: This affects all targets in the CMake scope where ```alpaka_RELOCATABLE_DEVICE_CODE``` is set. For the effects on SYCL code see Intel's documentation:

<https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-2/fsycl-rtc.html>

COMPILER SPECIFICS

Alpaka supports a large number of different compilers. Each of the compilers has its own special features. This page explains some of these specifics.

10.1 Choosing the correct Standard Library in Clang

Clang supports both, `libstdc++` shipped with the GNU GCC toolchain and `libc++`, LLVM's own implementation of the C++ standard library. By default, clang and all clang-based compilers, such as the `hipcc`, use `libstdc++` (GNU GCC). If more than one GCC version is installed, it is not entirely clear which version of `libstdc++` is selected. The following code can be used to check which standard library and version clang is using by default with the current setup.

```
#include <iostream>

int main(){
    #ifdef _GLIBCXX_RELEASE
        std::cout << "use libstdc++ (GNU GCC's standard library implementation)" <<
std::endl;
        std::cout << "version: " << _GLIBCXX_RELEASE << std::endl;
    #endif

    #ifdef _LIBCPP_VERSION
        std::cout << "use libc++ (LLVM's standard library implementation)" << std::endl;
        std::cout << "version: " << _LIBCPP_VERSION << std::endl;
    #endif
}
```

The command `clang -v ...` shows the include paths and also gives information about the standard library used.

10.1.1 Choose a specific `libstdc++` version

Clang provides the argument `--gcc-toolchain=<path>` which allows you to select the path of a GCC installation. For example, if you built the GCC compiler from source, you can select the installation prefix, which is the base folder with the subfolders `include`, `lib` and so on.

If you are using CMake, you can set the `--gcc-toolchain` flag via the following CMake command line argument:

- `-DCMAKE_CXX_FLAGS="--gcc-toolchain=<path>"` if you use Clang as compiler for CPU backends or the HIP backend.
- `-DCMAKE_CUDA_FLAGS="--gcc-toolchain=<path>"` if you use Clang as CUDA compiler.

Hint: If you are using Ubuntu and install a new gcc version via `apt`, it is not possible to select a specific gcc version because `apt` installs all headers and shared libraries in subfolders of `/usr/include` and `/usr/lib`. Therefore,

you can only use the /usr base path and Clang will automatically select one of the installed libstdc++ versions.

Hint: If you installed Clang/LLVM with spack and a gcc compiler, the Clang compiler will use the libstdc++ of the compiler used to build Clang/LLVM.

10.1.2 Selecting libc++

libc++ can be used if you set the compiler flag `-stdlib=libc++`.

If you are using CMake, you can select libc++ via the following CMake command line argument:

- `-DCMAKE_CXX_FLAGS="-stdlib=libc++"` if you use Clang as compiler for CPU backends or the HIP backend.
- `-DCMAKE_CUDA_FLAGS="-stdlib=libc++"` if you use Clang as CUDA compiler.

SIMILAR PROJECTS

11.1 KOKKOS

See also:

- <https://www.xsede.org/documents/271087/586927/Edwards-2013-XSCALE13-Kokkos.pdf>
- https://trilinos.org/oldsite/events/trilinos_user_group_2013/presentations/2013-11-TUG-Kokkos-Tutorial.pdf
- https://on-demand.gputechconf.com/supercomputing/2013/presentation/SC3103_Towards-Performance-Portable-Applications-Kokkos.pdf
- <https://dx.doi.org/10.3233/SPR-2012-0343>

Kokkos provides an abstract interface for portable, performant shared memory-programming. It is a C++ library that offers `parallel_for`, `parallel_reduce` and similar functions for describing the pattern of the parallel tasks. The execution policy determines how the threads are executed. For example, this influences the sizes of blocks of threads or if static or dynamic scheduling should be used. The library abstracts the kernel as a function object that can not have any user defined parameters for its `operator()`. Arguments have to be stored in members of the function object coupling algorithm and data together. *KOKKOS* provides both, abstractions for parallel execution of code and data management. Multidimensional arrays with a neutral indexing and an architecture dependent layout are available, which can be used, for example, to abstract the underlying hardware's preferred memory access scheme that could be row-major, column-major or even blocked.

11.2 Thrust

Thrust is a parallel algorithms library resembling the C++ Standard Template Library (STL). It allows to select either the *CUDA*, *TBB* or *OpenMP* back-end at make-time. Because it is based on generic `host_vector` and `device_vector` container objects, it is tightly coupling the data structure and the parallelization strategy. There exist many similar libraries such as *ArrayFire* (*CUDA*, *OpenCL*, native C++), *VexCL* (*OpenCL*, *CUDA*), *ViennaCL* (*OpenCL*, *CUDA*, *OpenMP*) and *hemi* (*CUDA*, native C++).

See also:

- Phalanx See [here](#) It is very similar to *alpaka* in the way it abstracts the accelerators. C++ Interface provides *CUDA*, *OpenMP*, and *GASNet* back-ends
- Aura
- Intel TBB
- UPC++

12.1 Accelerator Implementations

The table shows which native implementation or information is used to represent an alpaka functionality.

alpaka	Serial	std::thread	OpenMP 2.0	OpenMP 4.0	CUDA 9.0+
Devices	Host Core	Host Cores	Host Cores	Host Cores	NVIDIA GPUs
Lib/API	Standard C++	std::thread	OpenMP 2.0	OpenMP 4.0	CUDA 9.0+
Kernel execution	sequential	std::thread(kernel)	omp_set_dynamic(#pragma omp parallel num_threads(iNum	#pragma omp tar- get, #pragma omp teams num_teams(...) thread_limit(...), #pragma omp distribute, #pragma omp parallel num_threads(...)	cuda- Config- ureCall, cudaSe- tupAr- gument, cud- aLaunch
Execution strategy grid-blocks	sequential	sequential	sequential	undefined	undefined
Execution strategy block-kernels	sequential	preemptive multitasking	preemptive mul- titasking	preemptive multitasking	lock-step within warps
getIdx	emu- lated	block-kernel: mapping of std::this_thread:: grid-block: member vari- able	block-kernel: omp_get_num_thre to 3D index map- ping grid-block: member variable	block-kernel: omp_get_num_threads() to 3D index mapping grid- block: member variable	threadIdx, blockIdx
getExtents	member vari- ables	member vari- ables	member vari- ables	member variables	gridDim, blockDim
getBlock- SharedMem- DynSizeBytes	allo- cated in memory prior to kernel execu- tion	allocated in memory prior to kernel exe- cution	allocated in memory prior to kernel execution	allocated in memory prior to kernel execution	__shared__
allocBlock- SharedMem	master thread allocates	syncBlockKer- nels -> master thread allocates -> syncBlock- Kernels	syncBlockKer- nels -> master thread allocates -> syncBlock- Kernels	syncBlockKernels -> mas- ter thread allocates -> syncBlockKernels	__shared__
syncBlock- Kernels	not re- quired	barrier	#pragma omp barrier	#pragma omp barrier	__sync- threads
atomicOp	hier- archy de- pended	std::lock_guard< std::mutex >	#pragma omp critical	#pragma omp critical	atom- icXXX
AL- PAKA_FN_HO AL- PAKA_FN_AC AL- PAKA_FN_HO	inline	inline	inline	inline	__de- vice__, __host__, __forcein- line__

12.2 Serial

The serial accelerator only allows blocks with exactly one thread. Therefore it does not implement real synchronization or atomic primitives.

12.3 Threads

12.3.1 Execution

To prevent recreation of the threads between execution of different blocks in the grid, the threads are stored inside a thread pool. This thread pool is local to the invocation because making it local to the `KernelExecutor` could mean a heavy memory usage and lots of idling kernel-threads when there are multiple `KernelExecutors` around. Because the default policy of the threads in the pool is to yield instead of waiting, this would also slow down the system immensely.

12.4 OpenMP

12.4.1 Execution

Parallel execution of the kernels in a block is required because when `syncBlockThreads` is called all of them have to be done with their work up to this line. So we have to spawn one real thread per kernel in a block. `omp for` is not useful because it is meant for cases where multiple iterations are executed by one thread but in our case a 1:1 mapping is required. Therefore we use `omp parallel` with the specified number of threads in a block. Another reason for not using `omp for` like `#pragma omp parallel for collapse(3) num_threads(blockDim.x*blockDim.y*blockDim.z)` is that `#pragma omp barrier` used for intra block synchronization is not allowed inside `omp for` blocks.

Because OpenMP is designed for a 1:1 abstraction of hardware to software threads, the block size is restricted by the number of OpenMP threads allowed by the runtime. This could be as little as 2 or 4 kernels but on a system with 4 cores and hyper-threading OpenMP can also allow 64 threads.

12.4.2 Index

OpenMP only provides a linear thread index. This index is converted to a 3 dimensional index at runtime.

12.4.3 Atomic

We can not use `#pragma omp atomic` because braces or calling other functions directly after `#pragma omp atomic` are not allowed. Because we are implementing the CUDA atomic operations which return the old value, this requires `#pragma omp critical` to be used. `omp_set_lock` is an alternative but is usually slower.

12.5 CUDA

Nearly all CUDA functionality can be directly mapped to alpaka function calls. A major difference is that CUDA requires the block and grid sizes to be given in (x, y, z) order. alpaka uses the mathematical C/C++ array indexing scheme [z][y][x]. In both cases x is the innermost / fast running index.

Furthermore alpaka does not require the indices and extents to be 3-dimensional. The accelerators are templated on and support arbitrary dimensionality. NOTE: Currently the CUDA implementation is restricted to a maximum of 3 dimensions!

NOTE: You have to be careful when mixing alpaka and non alpaka CUDA code. The CUDA-accelerator back-end can change the current CUDA device and will NOT set the device back to the one prior to the invocation of the alpaka function.

Function Attributes

Depending on the cmake argument `ALPAKA_ACC_GPU_CUDA_ONLY_MODE` the function attributes are defined differently.

`ALPAKA_ACC_GPU_CUDA_ONLY_MODE=OFF` (default)

CUDA	alpaka
<code>__host__</code>	<code>ALPAKA_FN_HOST</code>
<code>__device__</code>	–
<code>__global__</code>	–
<code>__host__ __device__</code>	<code>ALPAKA_FN_HOST_ACC</code> , <code>ALPAKA_FN_ACC</code>

`ALPAKA_ACC_GPU_CUDA_ONLY_MODE=ON`

CUDA	alpaka
<code>__host__</code>	<code>ALPAKA_FN_HOST</code>
<code>__device__</code>	<code>ALPAKA_FN_ACC</code>
<code>__global__</code>	–
<code>__host__ __device__</code>	<code>ALPAKA_FN_HOST_ACC</code>

Note: There is no alpaka equivalent to `__global__` because the design of alpaka does not allow it. When running a alpaka kernel, alpaka creates a `__global__` kernel that performs some setup functions, such as creating the acc object, and then runs the user kernel, which must be a CUDA `__device__` function.

Note: You can not call CUDA-only methods, except when `ALPAKA_ACC_GPU_CUDA_ONLY_MODE` is enabled.

Note: When calling a `constexpr` function from inside a device function, also mark the called function as a device function, e.g. by prepending `ALPAKA_FN_ACC`.

Note that some compilers do that by default, but not all. For details please refer to [#1580](#).

Memory

CUDA	alpaka
<code>__shared__</code>	<code>alpaka::declareSharedVar<std::uint32_t, __COUNTER__>(acc)</code>
<code>__constant__</code>	<code>ALPAKA_STATIC_ACC_MEM_CONSTANT</code>
<code>__device__</code>	<code>ALPAKA_STATIC_ACC_MEM_GLOBAL</code>


```
template<typename T, std::size_t TuniqueId, typename TBlockSharedMemSt>
ALPAKA_NO_HOST_ACC_WARNING ALPAKA_FN_ACC auto alpaka::declareSharedVar(TBlockSharedMemSt
                                                                    const
                                                                    &block-
                                                                    Shared-
                                                                    MemSt) ->
                                                                    T&
```

Declare a block shared variable.

The variable is uninitialized and not default constructed! The variable can be accessed by all threads within a block. Access to the variable is not thread safe.

Template Parameters

- **T** – The element type.
- **TuniqueId** – id those is unique inside a kernel
- **TBlockSharedMemSt** – The block shared allocator implementation type.

Parameters

blockSharedMemSt – The block shared allocator implementation.

Returns

Uninitialized variable stored in shared memory.

ALPAKA_STATIC_ACC_MEM_CONSTANT

This macro defines a variable lying in constant accelerator device memory.

Example: `ALPAKA_STATIC_ACC_MEM_CONSTANT int i;`

Those variables behave like ordinary variables when used in file-scope. They have external linkage (are accessible from other compilation units). If you want to access it from a different compilation unit, you have to declare it as extern: `extern ALPAKA_STATIC_ACC_MEM_CONSTANT int i;` Like ordinary variables, only one definition is allowed (ODR) Failure to do so might lead to linker errors.

In contrast to ordinary variables, you can not define such variables as static compilation unit local variables with internal linkage because this is forbidden by CUDA.

Attention

It is not allowed to initialize the variable together with the declaration. To initialize the variable `alpaka::createStaticDevMemView` and `alpaka::memcpy` must be used.

```
ALPAKA_STATIC_ACC_MEM_CONSTANT int foo;

void initFoo() {
    auto extent = alpaka::Vec<alpaka::DimInt<1u>, size_t>{1};
    auto viewFoo = alpaka::createStaticDevMemView(&foo, device, extent);
    int initialValue = 42;
    alpaka::ViewPlainPtr<DevHost, int, alpaka::DimInt<1u>, size_t> bufHost(&
↪initialValue, devHost, extent);
    alpaka::memcpy(queue, viewGlobalMemUninitialized, bufHost, extent);
}
```

ALPAKA_STATIC_ACC_MEM_GLOBAL

This macro defines a variable lying in global accelerator device memory.

Example: `ALPAKA_STATIC_ACC_MEM_GLOBAL int i;`

Those variables behave like ordinary variables when used in file-scope. They have external linkage (are accessible from other compilation units). If you want to access it from a different compilation unit, you have

to declare it as extern: `extern ALPAKA_STATIC_ACC_MEM_GLOBAL int i;` Like ordinary variables, only one definition is allowed (ODR) Failure to do so might lead to linker errors.

In contrast to ordinary variables, you can not define such variables as static compilation unit local variables with internal linkage because this is forbidden by CUDA.

Attention

It is not allowed to initialize the variable together with the declaration. To initialize the variable `alpaka::createStaticDevMemView` and `alpaka::memcpy` must be used.

```
ALPAKA_STATIC_ACC_MEM_GLOBAL int foo;

void initFoo() {
    auto extent = alpaka::Vec<alpaka::DimInt<1u>, size_t>{1};
    auto viewFoo = alpaka::createStaticDevMemView(&foo, device, extent);
    int initialValue = 42;
    alpaka::ViewPlainPtr<DevHost, int, alpaka::DimInt<1u>, size_t> bufHost(&
    ↪initialValue, devHost, extent);
    alpaka::memcpy(queue, viewGlobalMemUninitialized, bufHost, extent);
}
```

Index / Work Division

CUDA	alpaka
<code>threadIdx</code>	<code>alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc)</code>
<code>blockIdx</code>	<code>alpaka::getIdx<alpaka::Grid, alpaka::Blocks>(acc)</code>
<code>blockDim</code>	<code>alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(acc)</code>
<code>gridDim</code>	<code>alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(acc)</code>
<code>warpSize</code>	<code>alpaka::warp::getSize(acc)</code>

Types

12.5.1 CUDA Runtime API

The following tables list the functions available in the [CUDA Runtime API](#) and their equivalent alpaka functions:

Device Management

CUDA	alpaka
cudaChooseDevice	—
cudaDeviceGetAttribute	—
cudaDeviceGetByPCIBusId	—
cudaDeviceGetCacheConfig	—
cudaDeviceGetLimit	—
cudaDeviceGetP2PAttribute	—
cudaDeviceGetPCIBusId	—
cudaDeviceGetSharedMemConfig	—
cudaDeviceGetQueuePriorityRange	—
cudaDeviceReset	alpaka::reset(device)
cudaDeviceSetCacheConfig	—
cudaDeviceSetLimit	—
cudaDeviceSetSharedMemConfig	—
cudaDeviceSynchronize	void alpaka::wait(device)
cudaGetDevice	n/a (no current device)
cudaGetDeviceCount	std::size_t alpaka::getDevCount< TPlatform >()
cudaGetDeviceFlags	—
cudaGetDeviceProperties	alpaka::getAccDevProps(dev) (Only some properties available)
cudaIpcCloseMemHandle	—
cudaIpcGetEventHandle	—
cudaIpcGetMemHandle	—
cudaIpcOpenEventHandle	—
cudaIpcOpenMemHandle	—
cudaSetDevice	n/a (no current device)
cudaSetDeviceFlags	—
cudaSetValidDevices	—

Error Handling

CUDA	alpaka
cudaGetErrorName	n/a (handled internally, available in exception message)
cudaGetErrorString	n/a (handled internally, available in exception message)
cudaGetLastError	n/a (handled internally)
cudaPeekAtLastError	n/a (handled internally)

Queue Management

CUDA	alpaka
cudaLaunchHostFunc cudaStreamAddCallback	alpaka::enqueue(queue, [](){dosomething();})
cudaStreamAttachMemAsync cudaStreamCreate	– • queue=alpaka::QueueCudaRtNonBlocking(device); • queue=alpaka::QueueCudaRtBlocking(device);
cudaStreamCreateWithFlags	see cudaStreamCreate (cudaStreamNonBlocking hard coded)
cudaStreamCreateWithPriority	–
cudaStreamDestroy	n/a (Destructor)
cudaStreamGetFlags	–
cudaStreamGetPriority	–
cudaStreamQuery	bool alpaka::empty(queue)
cudaStreamSynchronize	void alpaka::wait(queue)
cudaStreamWaitEvent	void alpaka::wait(queue, event)

Event Management

CUDA	alpaka
cudaEventCreate	alpaka::Event< TQueue > event(dev);
cudaEventCreateWithFlags	–
cudaEventDestroy	n/a (Destructor)
cudaEventElapsedTime	–
cudaEventQuery	bool alpaka::isComplete(event)
cudaEventRecord	void alpaka::enqueue(queue, event)
cudaEventSynchronize	void alpaka::wait(event)

*Memory Management**Execution Control*

CUDA	alpaka
cudaFuncGetAttributes	–
cudaFuncSetCacheConfig	–
cudaFuncSetSharedMemConfig	–
cudaLaunchKernel	• alpaka::exec<TAcc>(queue, workDiv, kernel, params...) • auto byteDynSharedMem = alpaka::getBlockSharedMemDynSizeBytes(kernel, ...)
cudaSetDoubleForDevice	n/a (alpaka assumes double support)
cudaSetDoubleForHost	n/a (alpaka assumes double support)

Occupancy

CUDA	alpaka
cudaOccupancyMaxActiveBlocksPerMultiprocessor	–
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags	–

Unified Addressing

CUDA	alpaka
cudaPointerGetAttributes	–

Peer Device Memory Access

CUDA	alpaka
cudaDeviceCanAccessPeer	–
cudaDeviceDisablePeerAccess	–
cudaDeviceEnablePeerAccess	automatically done when required

OpenGL, Direct3D, VDPAU, EGL, Graphics Interoperability*not available***Texture/Surface Reference/Object Management***not available***Version Management***not available*

12.6 HIP

Warning: The HIP documentation is outdated and must be overworked.

12.6.1 Current Restrictions on HCC platform

- Workaround for unsupported `syncthreads_{count|and|or}`.
 - Uses temporary shared value and atomics
- Workaround for buggy `hipStreamQuery`, `hipStreamSynchronize`.
 - Introduces own queue management
 - `hipStreamQuery` and `hipStreamSynchronize` do not work in multithreaded environment
- Workaround for missing `cuStreamWaitValue32`.
 - Polls value each 10 ms
- Device constant memory not supported yet
- Note that `printf` in kernels is still not supported in HIP
- Exclude `hipMalloc3D` and `hipMallocPitch` when size is zero otherwise they throw an Unknown Error
- `TestAccs` excludes 3D specialization of HIP back-end for now because `verifyBytesSet` fails in `memView` for 3D specialization
- `dim3` structure is not available on device (use `alpaka::Vec` instead)
- Constructors' attributes unified with destructors'.
 - Host/device signature must match in HIP(HCC)
- A chain of functions must also provide correct host-device signatures
 - E.g. a host function cannot be called from a host-device function

- Recompile your target when HCC linker returned the error: “File format not recognized clang-7: error: linker command failed with exit code 1”
- If compile-error occurred the linker still may link, but without the device code
- AMD device architecture currently hardcoded in `alpakaConfig.cmake`

12.6.2 Compiling HIP from Source

Follow [HIP Installation](#) guide for installing HIP. HIP requires either `nvcc` or `hcc` to be installed on your system (see guide for further details).

- If you want the HIP binaries to be located in a directory that does not require superuser access, be sure to change the install directory of HIP by modifying the `CMAKE_INSTALL_PREFIX` cmake variable.
- Also, after the installation is complete, add the following line to the `.profile` file in your home directory, in order to add the path to the HIP binaries to `PATH`: `PATH=$PATH:<path_to_binaries>`

```
git clone --recursive https://github.com/ROCm-Developer-Tools/HIP.git
cd HIP
mkdir -p build
cd build
cmake -DCMAKE_BUILD_TYPE="${CMAKE_BUILD_TYPE}" -DCMAKE_INSTALL_PREFIX=${YOUR_HIP_
↳INSTALL_DIR} -DBUILD_TESTING=OFF ..
make
make install
```

- Set the appropriate paths (edit `${YOUR_**}` variables)

```
# HIP_PATH required by HIP tools
export HIP_PATH=${YOUR_HIP_INSTALL_DIR}
# Paths required by HIP tools
export CUDA_PATH=${YOUR_CUDA_ROOT}
# - if required, path to HCC compiler. Default /opt/rocm/hcc.
export HCC_HOME=${YOUR_HCC_ROOT}
# - if required, path to HSA include, lib. Default /opt/rocm/hsa.
export HSA_PATH=${YOUR_HSA_PATH}
# HIP binaries and libraries
export PATH=${HIP_PATH}/bin:$PATH
export LD_LIBRARY_PATH=${HIP_PATH}/lib64:${LD_LIBRARY_PATH}
```

- Test the HIP binaries

```
# calls nvcc or hcc
which hipcc
hipcc -V
which hipconfig
hipconfig -v
```

12.6.3 Verifying HIP Installation

- If PATH points to the location of the HIP binaries, the following command should list several relevant environment variables, and also the selected compiler on your system-`\`hipconfig -f\``
- Compile and run the [square sample](#), as pointed out in the original [HIP install guide](#).

12.6.4 Compiling Examples with HIP Back End

As of now, the back-end has only been tested on the NVIDIA platform.

- NVIDIA Platform
 - One issue in this branch of alpaka is that the host compiler flags don't propagate to the device compiler, as they do in CUDA. This is because a counterpart to the `CUDA_PROPAGATE_HOST_FLAGS` cmake variable has not been defined in the `FindHIP.cmake` file. alpaka forwards the host compiler flags in cmake to the `HIP_NVCC_FLAGS` cmake variable, which also takes user-given flags. To add flags to this variable, toggle the advanced mode in `ccmake`.

12.6.5 Random Number Generator Library rocRAND for HIP Back End

rocRAND provides an interface for HIP, where the `cuRAND` or `rocRAND` API is called depending on the chosen HIP platform (can be configured with cmake in alpaka).

Clone the *rocRAND* repository, then build and install it

```
git clone https://github.com/ROCmSoftwarePlatform/rocRAND
cd rocRAND
mkdir -p build
cd build
cmake -DCMAKE_INSTALL_PREFIX=${HIP_PATH} -DBUILD_BENCHMARK=OFF -DBUILD_TEST=OFF -
↳DCMAKE_MODULE_PATH=${HIP_PATH}/cmake . .
make
```

The `CMAKE_MODULE_PATH` is a cmake variable for locating module finding scripts like *FindHIP.cmake*. The paths to the *rocRAND* library and include directories should be appended to the `CMAKE_PREFIX_PATH` variable.

DETAILS

Execution Domain	Code		Concepts	Implementations									
Host	alpaka	User	Device Manager	PltfCpu						PltfCudaRt			
			Device Waitable	DevCpu						DevCudaRt			
			Queue Waitable	QueueCpuNonBlocking			QueueCpuBlocking			QueueCudaRtNonBlocking	QueueCudaRtBlocking		
			Event Waitable	EventCpu						EventCudaRt			
			Buffer View Offset Extent Dim Idx	BufCpu						BufCudaRt			
			std::vector										
			std::array										
			TaskKernel Dim Idx	TaskKernelCpuSerial	TaskKernelCpuFibers	TaskKernelCpuOmp2Blocks	TaskKernelCpuOmp2Threads	TaskKernelCpuThreads	TaskKernelCpuOmp4	TaskKernelGpuCudaRt			
Accelerator	User	Kernel	...										
	alpaka	User	Accelerator Dim Idx	AccCpuSerial	AccCpuOmp2Blocks	AccCpuFibers	AccCpuOmp2Threads	AccCpuOmp4	AccCpuThreads	AccGpuCudaRt			
			Work Division	WorkDivMember						WorkDivCudaBuiltIn			
			Index	IdxGbRef						IdxGbCudaBuiltIn			
				IdxBtZero	IdxBtFiberIdMap	IdxBtOmp		IdxBtThreadIdMap		IdxBtCudaBuiltIn			
			Atomic	AtomicNoOp			AtomicOmpCriticalSection		AtomicStdLibLock	AtomicCudaBuiltIn			
			Rand	RandStdLib						RandCuRand			
			Math	MathStdLib						MathCudaBuiltIn			
			...										

The full stack of concepts defined by the *alpaka* library and their inheritance hierarchy is shown in the third column of the preceding figure. Default implementations for those concepts can be seen in the blueish columns. The various accelerator implementations, shown in the lower half of the figure, only differ in some of their underlying concepts but can share most of the base implementations. The default implementations can, but do not have to be used at all. They can be replaced by user code in arbitrary granularity. By substituting, for instance, the atomic operation implementation of an accelerator, the execution can be fine-tuned, to better utilize the hardware instruction set of a specific processor. However, also complete accelerators, devices and all of the other concepts can be implemented by the user without the need to change any part of the *alpaka* library itself. The way this and other things are implemented is explained in the following paragraphs.

13.1 Concept Implementations

The *alpaka* library has been implemented with extensibility in mind. This means that there are no predefined classes, modeling the concepts, the *alpaka* functions require as input parameters. They allow arbitrary types as parameters, as long as they model the required concept.

C++ provides a language inherent object oriented abstraction allowing to check that parameters to a function comply with the concept they are required to model. By defining interface classes, which model the *alpaka* concepts, the user would be able to inherit his extension classes from the interfaces he wants to model and implement the abstract virtual methods the interfaces define. The *alpaka* functions in turn would use the corresponding interface types as their parameter types. For example, the *Buffer* concept requires methods for getting the pitch or accessing the underlying memory. With this intrusive object oriented design pattern the *BufCpu* or *BufCudaRt* classes would have to inherit from an *IBuffer* interface and implement the abstract methods it declares. An example of this basic pattern is shown in the following source snippet:

```
struct IBuffer
{
    virtual std::size_t getPitch() const = 0;
    virtual std::byte * data() = 0;
    ...
};

struct BufCpu : public IBuffer
{
    virtual std::size_t getPitch() const override { ... }
    virtual std::byte * data() override { ... }
    ...
};

ALPAKA_FN_HOST auto copy(
    IBuffer & dst,
    IBuffer const & src)
-> void
{
    ...
}
```

The compiler can then check at compile time that the objects the user wants to use as function parameters can be implicitly cast to the interface type, which is the case for inherited base classes. The compiler returns an error message on a type mismatch. However, if the *alpaka* library were using those language inherent object oriented abstractions, the extensibility and optimizability it promises would not be possible. Classes and run-time polymorphism require the implementer of extensions to intrusively inherit from predefined interfaces and override special virtual functions.

This is feasible for user defined classes or types where the source code is available and where it can be changed. The `std::vector` class template on the other hand would not be able to model the *Buffer* concept because we can not change its definition to inherit from the *IBuffer* interface class since it is part of the standard library. The standard inheritance based object orientation of C++ only works well when all the code it is to interoperate with can be changed to implement the interfaces. It does not enable interaction with unalterable or existing code that is too complex to change, which is the reality in the majority of software projects.

Another option to implement an extensible library is to follow the way the C++ standard library uses. It allows to specialize function templates for user types to model concepts without altering the types themselves. For example, the `std::begin` and `std::end` free function templates can be specialized for user defined types. With those functions specialized, the C++11 range-based for loops (`for(auto & i : userContainer){...}`) see C++ *Standard 6.5.4/1* can be used with user defined types. Equally specializations of `std::swap` and other standard library function templates can be defined to extend those with support for user types. One Problem with function specialization is, that only full specializations are allowed. A partial function template specialization is not allowed by the standard. Another problem can emerge due to users carelessly overloading the template functions instead

of specializing them. Mixing function overloading and function template specialization on the same base template function can result in unexpected results. The reasons and effects of this are described more closely in an article from H. Sutter (currently convener of the ISO C++ committee) called *Sutter's Mill: Why Not Specialize Function Templates?* in the *C/C++ Users Journal* in July 2001.

See also:

[different way](#)

The solution given in the article is to provide “a single function template that should never be specialized or overloaded”. This function simply forwards its arguments “to a class template containing a static function with the same signature”. This template class can fully or partially be specialized without affecting overload resolution.

The way the *alpaka* library implements this is by not using the C++ inherent object orientation but lifting those abstractions to a higher level. Instead of using a non-extensible “class”/struct and abstract virtual member functions for the interface, *alpaka* defines free functions. All those functions are templates allowing the user to call them with arbitrary self defined types and not only those inheriting from a special interface type. Unlike member functions, they have no implicit `this` pointer, so the object instance has to be explicitly given as a parameter. Overriding the abstract virtual interface methods is replaced by the specialization of a template type that is defined for each such function.

A concept is completely implemented by specializing the predefined template types. This allows to extend and fine-tune the implementation non-intrusively. For example, the corresponding pitch and memory pinning template types can be specialized for `std::vector`. After doing this, the `std::vector` can be used everywhere a buffer is accepted as argument throughout the whole *alpaka* library without ever touching its definition.

A simple function allowing arbitrary tasks to be enqueued into a queue can be implemented in the way shown in the following code. The `TSfinae` template parameter will be explained in a *following section*.

```
namespace alpaka
{
    template<
        typename TQueue,
        typename TTask,
        typename TSfinae = void>
        struct Enqueue;

    template<
        typename TQueue,
        typename TTask>
    ALPAKA_FN_HOST auto enqueue(
        TQueue & queue,
        TTask & task)
    -> void
    {
        Enqueue<
            TQueue,
            TTask>
            ::enqueue(
                queue,
                task);
    }
}
```

A user who wants his queue type to be used with this `enqueue` function has to specialize the `Enqueue` template struct. This can be either done partially by only replacing the `TQueue` template parameter and accepting arbitrary tasks or by fully specializing and replacing both `TQueue` and `TTask`. This gives the user complete freedom of choice. The example given in the following code shows this by specializing the `Enqueue` type for a user queue type `UserQueue` and arbitrary tasks.

```
struct UserQueue{};

namespace alpaka
{
    // partial specialization
    template<
        typename TTask>
        struct Enqueue<
            UserQueue
            TTask>
        {
            ALPAKA_FN_HOST static auto enqueue(
                UserQueue & queue,
                TTask & task)
            -> void
            {
                //...
            }
        };
}
```

In addition the subsequent code shows a full specialization of the Enqueue type for a given UserQueue and a UserTask.

```
struct UserQueue{};
struct UserTask{};

namespace alpaka
{
    // full specialization
    template<>
    struct Enqueue<
        UserQueue
        UserTask>
    {
        ALPAKA_FN_HOST static auto enqueue(
            UserQueue & queue,
            UserTask & task)
        -> void
        {
            //...
        }
    };
}
```

When the enqueue function template is called with an instance of UserQueue, the most specialized version of the Enqueue template is selected depending on the type of the task TTask it is called with.

A type can model the queue concept completely by defining specializations for `alpaka::Enqueue` and `alpaka::Empty`. This functionality can be accessed by the corresponding `alpaka::enqueue` and `alpaka::empty` template functions.

Currently there is no native language support for describing and checking concepts in C++ at compile time. A study group (SG8) is working on the ISO [specification for concepts](#) and compiler forks implementing them do exist. For usage in current C++ there are libraries like `Boost::ConceptCheck` which try to emulate requirement checking of concept types. Those libraries often exploit the preprocessor and require non-trivial changes to the function declaration syntax. Therefore the *alpaka* library does not currently make use of *Boost::ConceptCheck*. Neither does it facilitate the proposed concept specification due to its dependency on non-standard compilers.

The usage of concepts as described in the working draft would often dramatically enhance the compiler error messages in case of violation of concept requirements. Currently the error messages are pointing deeply inside the stack of library template invocations where the missing method or the like is called. Instead of this, with concept checking it would directly fail at the point of invocation of the outermost template function with an expressive error message about the parameter and its violation of the concept requirements. This would simplify especially the work with extendable template libraries like *Boost* or *alpaka*. However, in the way concept checking would be used in the *alpaka* library, omitting it does not change the semantic of the program, only the compile time error diagnostics. In the future when the standard incorporates concept checking and the major compilers support it, it will be added to the *alpaka* library.

13.2 Template Specialization Selection on Arbitrary Conditions

Basic template specialization only allows for a selection of the most specialized version where all explicitly stated types have to be matched identically. It is not possible to enable or disable a specialization based on arbitrary compile time expressions depending on the parameter types. To allow such conditions, *alpaka* adds a defaulted and unused TSfinae template parameter to all declarations of the implementation template structs. This was shown using the example of the Enqueue template type. The C++ technique called SFINAE, an acronym for *Substitution failure is not an error* allows to disable arbitrary specializations depending on compile time conditions. Specializations where the substitution of the parameter types by the deduced types would result in invalid code will not result in a compile error, but will simply be omitted. An example in the context of the Enqueue template type is shown in the following code.

```
struct UserQueue{};

namespace alpaka
{
    template<
        typename TQueue,
        typename TTask>
    struct Enqueue<
        TQueue
        TTask,
        std::enable_if_t<
            std::is_base_of_v<UserQueue, TQueue>
            && (TTask::TaskId == 1u)
        >>
    {
        ALPAKA_FN_HOST static auto enqueue(
            TQueue & queue,
            TTask & task)
        -> void
        {
            //...
        }
    };
}
```

The Enqueue specialization shown here does not require any direct type match for the TQueue or the TTask template parameter. It will be used in all contexts where TQueue has inherited from UserQueue and where the TTask has a static const integral member value TaskId that equals one. If the TTask type does not have a TaskId member, this code would be invalid and the substitution would fail. However, due to SFINAE, this would not result in a compiler error but rather only in omitting this specialization. The `std::enable_if` template results in a valid expression, if the condition it contains evaluates to true, and an invalid expression if it is false. Therefore it can be used to disable specializations depending on arbitrary boolean conditions. It is utilized in the case where the TaskId member is unequal one or the TQueue does not inherit from UserQueue. In this circumstances, the condition itself results in valid code but because it evaluates to false, the `std::enable_if` specialization results in invalid code and the whole Enqueue template specialization gets omitted.

13.3 Argument dependent lookup for math functions

Alpaka comes with a set of basic mathematical functions in the namespace `alpaka::math`. These functions are dispatched in two ways to support user defined overloads of these functions.

Let's take `alpaka::math::abs` as an example: When `alpaka::math::abs(acc, value)` is called, a concrete implementation of `abs` is picked via template specialization. Concretely, something similar to `alpaka::math::trait::Abs<decltype(acc), decltype(value)>{}(acc, value)` is called. This allows alpaka (and the user) to specialize the template `alpaka::math::trait::Abs` for various backends and various argument types. E.g. alpaka contains specializations for `float` and `double`. If there is no specialization within alpaka (or by the user), the default implementation of `alpaka::math::trait::Abs<...>{}(acc, value)` will just call `abs(value)`. This is called an unqualified call and C++ will try to find a function called `abs` in the namespace where the type of `value` is defined. This feature is called Argument Dependent Lookup (ADL). Using ADL for types which are not covered by specializations in alpaka allows a user to bring their own implementation for which `abs` is meaningful, e.g. a custom implementation of complex numbers or a fixed precision type.

CODING GUIDELINES

Attention: The Coding Guidelines are currently revised

14.1 General

- Use the `.clang-format` file supplied in alpaka's top-level directory to format your code. This will handle indentation,

whitespace and braces automatically. Usage:

```
clang-format-16 -i <sourcefile>
```

- If you want to format the entire code base execute the following command from alpaka's top-level directory:

```
find example include test -name '*.hpp' -o -name '*.cpp' | xargs clang-format-16 -i
```

Windows users should use *Visual Studio's native clang-format integration* [<https://devblogs.microsoft.com/cppblog/clangformat-support-in-visual-studio-2017-15-7-preview-1/>](https://devblogs.microsoft.com/cppblog/clangformat-support-in-visual-studio-2017-15-7-preview-1/).

14.2 Naming

- Types are always in PascalCase (KernelExecCuda, BufT, ...) and singular.
- Variables are always in camelCase (memBufHost, ...) and plural for collections and singular else.
- Namespaces are always in lowercase and singular is preferred.
- There are no two consecutive upper case letters (AccOpenMp, HtmlRenderer, IoHandler, ...). This makes names more easily readable.

14.3 Types

- Always use integral types with known width (`int32_t`, `uint64_t`, ...). Never use `int`, `unsigned long`, etc.

14.4 Type Qualifiers

The order of type qualifiers should be: `Type const * const` for a const pointer to a const Type. `Type const &` for a reference to a const Type.

The reason is that types can be read from right to left correctly without jumping back and forth. `const Type * const` and `const Type &` would require jumping in either way to read them correctly.

14.5 Variables

- Variables should always be initialized on construction because this can produce hard to debug errors. This can (nearly) always be done even in performance critical code without sacrificing speed by using a functional programming style.
- Variables should (nearly) always be `const` to make the code more easy to understand. This is equivalent to functional programming and the SSA (static single assignment) style used by LLVM. This should have no speed implication as every half baked compiler analyses the usage of variables and reuses registers.
- Variable definitions should be differentiated from assignments by using either `(...)` or `{...}` but never `=` for definitions. Use `uint32_t const iUsageOfThisVariable(42);` instead of `uint32_t const iUsageOfThisVariable = 42;`

14.6 Comments

- Always use C++-Style comments `//`
- For types use `///#####` to start the comment block.
- For functions use `//-----` to start the comment block.

14.7 Functions

- Always use the trailing return type syntax with the return type on a new line even if the return type is void:

```
auto func()
-> bool
```

- This makes it easier to see the return type because it is on its own line.
- This leads to a consistent style for constructs where there is no alternative style (lambdas, functions templates with dependent return types) and standard functions.
- Each function parameter is on a new indented line:

```
auto func(
    float f1,
    float f2)
-> bool
{
    return true
}
```



```
func(
    1.0f,
    2.0f);
```

- Makes it easier to see how many parameters there are and which position they have.

14.8 Templates

- Template parameters are prefixed with T to differentiate them from class or function local typedefs.
- Each template parameter is on a new indented line:

```
template<
    typename TParam,
    typename TArgs...>
auto func()
-> bool
```

- Makes it easier to see how many template parameters there are and which position they have.
- Always use `typename` for template parameters. There is NO difference to class and `typename` matches the intent better.

14.9 Traits

- Trait classes always have one more template parameter (with default parameter) then is required for enabling SFINAE in the specialization:

```
template<
    typename T,
    typename TSfinae = void>
struct GetOffsets;
```

- Template trait aliases always end with a T e.g. `BufT` while the corresponding trait ends with `Type` e.g. `BufType`
- Traits for implementations always have the same name as the accessor function but in PascalCase while the member function is camelCase again: `sin(){...}` and `Sin{sin(){...}}`;

14.10 Includes

- The order of includes is from the most specialized header to the most general one. This order helps to find missing includes in more specialized headers because the general ones are always included afterwards.
- A comment with the types or functions included by a include file make it easier to find out why a special header is included.

Section author: Axel Huebl, alpaka-group

In the following section we explain how to contribute to this documentation.

If you are reading the [HTML version](#) and want to improve or correct existing pages, check the “[Edit on GitHub](#)” link on the right upper corner of each document.

Alternatively, go to *docs/source* in our source code and follow the directory structure of [reStructuredText](#) (*.rst*) files there. For intrusive changes, like structural changes to chapters, please open an issue to discuss them beforehand.

15.1 Build Locally

This document is build based on free open-source software, namely [Sphinx](#), [Doxygen](#) (C++ APIs as XML), [Breathe](#) (to include doxygen XML in Sphinx) and [rst2pdf](#) (render the cheat sheet). A web-version is hosted on [ReadTheDocs](#).

The following requirements need to be installed (once) to build our documentation successfully:

```
cd docs/

# doxygen is not shipped via pip, install it externally,
# from the homepage, your package manager, conda, etc.
# example:
sudo apt-get install doxygen
# sudo pacman -S doxygen

# python tools & style theme
pip install -r requirements.txt # --user
```

With all documentation-related software successfully installed, just run the following commands to build your docs locally. Please check your documentation build is successful and renders as you expected before opening a pull request!

```
# skip this if you are still in docs/
cd docs/

# parse the C++ API documentation (default: xml format)
doxygen Doxyfile

# render the cheatsheet.pdf
rst2pdf -s cheatsheet/cheatsheet.style source/basic/cheatsheet.rst -o cheatsheet/
↪ cheatsheet.pdf

# render the '.rst' files with sphinx
```

(continues on next page)

(continued from previous page)

```
make html

# open it, e.g. with firefox :)
firefox build/html/index.html

# now again for the pdf :)
make latexpdf

# open it, e.g. with okular
build/latex/alpaka.pdf
```

Hint: Run *make clean* to clean the build directory before executing actual make. This is necessary to reflect changes outside the rst files.

Hint: There is a `checklinks` target to check links in the rst files on availability:

```
# check existence of links
# cd docs/
make checklinks
```

Hint: The Doxyfile for doxygen is configured to output in xml format per default. Another targets can be configured in the Doxyfile. The final documentations are stored in `docs/doxygen/`.

```
# run in docs/doxygen/
sed -i -E 's/(GENERATE_HTML\s*=\s*)NO/\1YES/g' Doxyfile
```

15.2 readthedocs

To maintain or import a github project an account on [ReadTheDocs](#) is required. Further instructions can be found on [readthedocs on github](#) and [readthedocs import guide](#).

15.3 Useful Links

- [A primer on writing reStructuredText files for sphinx](#)
- [Why You Shouldn't Use "Markdown" for Documentation](#)
- [reStructuredText vs. Markdown](#)
- [Markdown Limitations in Sphinx](#)

AUTOMATIC TESTING

For automatic testing we use two different systems: GitHub Actions and GitLab CI. GitHub Actions are used for a wide range of build tests and also some CPU runtime tests. GitLab CI allows us to run runtime tests on GPUs and CPU architectures other than x86, like ARM or IBM POWER.

16.1 GitHub Actions

The configuration of GitHub Actions can be found in the `.github/workflows/` folder. This CI uses unmodified containers from Docker Hub and sets up the environment during the test job. A caching mechanism speeds up the job times. The scripts for setting up the environment, building alpaka and running test are located in the `script/` folder.

16.1.1 clang-format

The first CI job run is clang-format, which will verify the formatting of your changeset. Only if this check passes, will the remainder of the GitHub CI continue. In case of a formatting failure, a patch file is attached as an artifact to the GitHub action run. You can apply this patch file to your changeset to fix the formatting.

16.2 GitLab CI

We use GitLab CI because it allows us to use self-hosted system, e.g. GPU systems. The GitHub repository is mirrored on <https://gitlab.com/hzdr/crp/alpaka>. Every commit or pull request is automatically mirrored to GitLab and triggers the CI. The configuration of the GitLab CI is stored in the file `.gitlab-ci.yml`. The workflow of a GitLab CI is different from GitHub Actions. Instead of downloading an unmodified container from Docker Hub and preparing the environment during the test job, GitLab CI uses containers which are already prepared for the tests. The containers are built in an [extra repository](#) and contain all dependencies for alpaka. All available containers can be found [here](#). The scripts to build alpaka and run the tests are shared with GitHub Actions and located at `script/`.

Most of the jobs for the GitLab CI are generated automatically. For more information, see the section *The Job Generator*.

It is also possible to define custom jobs, see *Custom jobs*.

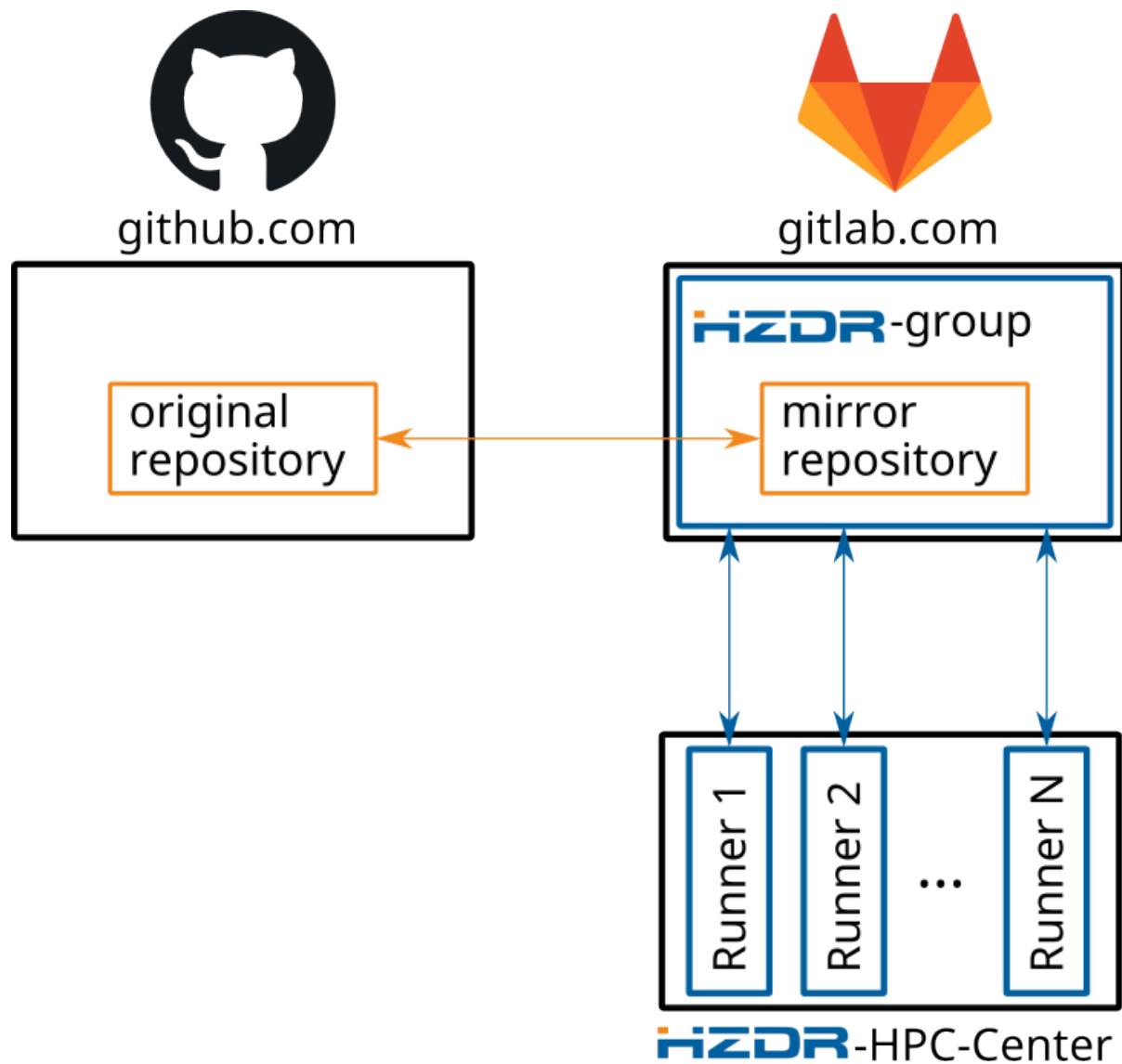


Fig. 1: Relationship between GitHub.com, GitLab.com and HZDR gitlab-ci runners

16.2.1 The Container Registry

Alpaka uses containers in which as many dependencies as possible are already installed to save job execution time. The available containers can be found [here](#). Each container provides a tool called `agc-manager` to check if a software is installed. The documentation for `agc-manager` can be found [here](#). A common way to check if a software is already installed is to use an `if else` statement. If a software is not installed yet, you can install it every time at job runtime.

```
if agc-manager -e boost@${ALPAKA_CI_BOOST_VER} ; then
    export ALPAKA_CI_BOOST_ROOT=$(agc-manager -b boost@${ALPAKA_CI_BOOST_VER})
else
    # install boost
fi
```

This statement installs a specific boost version until the boost version is pre-installed in the container. To install a specific software permanently in the container, please open an issue in the [alpaka-group-container repository](#).

16.2.2 The Job Generator

Alpaka supports a large number of different compilers with different versions and build configurations. To manage this large set of possible test cases, we use a job generator that generates the CI jobs for the different compiler and build configuration combinations. The jobs do not cover all possible combinations, as it would be too much to run the entire CI pipeline in a reasonable amount of time. Instead, the job generator uses [pairwise testing](#).

The stages of the job generator are:

The job generator is located at `script/job_generator/`. The code is split into two parts. One part is alpaka-specific and stored in this repository. The other part is valid for all alpaka-based projects and stored in the [alpaka-job-coverage library](#).

Run Job Generator Offline

First you need to install the dependencies. It is highly recommended to use a virtual environment. You can create one for example with the `venv`-Python module or with `miniconda`. Once you have created a virtual environment, you should activate it and install the Python packages via:

```
pip install -r script/job_generator/requirements.txt
```

After installing the Python package, you can simply run the job generator via:

```
# 3.0 is the version of the docker container image
# run `python ci/job_generator/job_generator.py --help` to see more options
python script/job_generator/job_generator.py 3.0
```

The generator creates a `jobs.yaml` in the current directory with all job combinations.

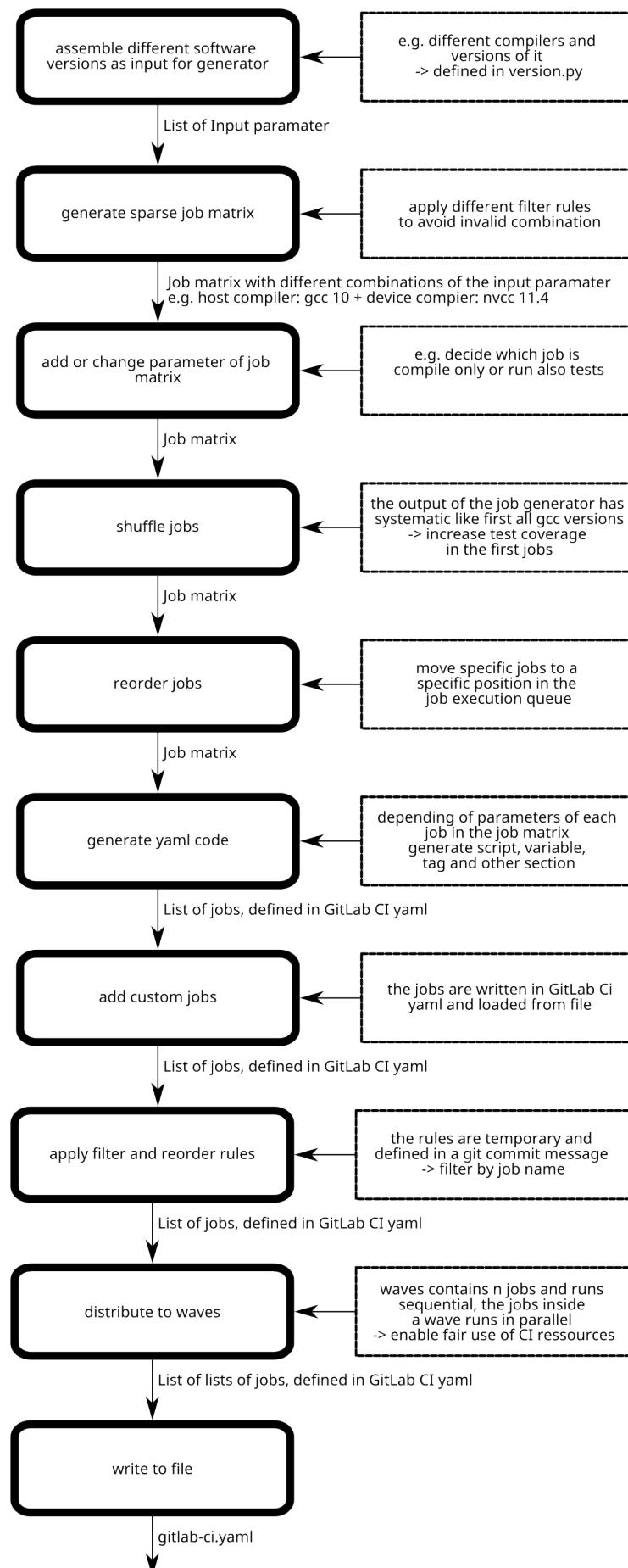
Filter and Reorder Jobs

The job generator provides the ability to filter and reorder the generated job matrix using [Python](#) regex. The regex is applied via the commit message for the current commit:

Add function to **filter and** reorder CI jobs

This commit message demonstrates how it works. The job **filter** removes **all** jobs whose names do **not** begin **with** NVCC **or** GCC. Then the jobs are reordered. First **all** GCC11 are executed, then **all** GCC8 **and** then the rest.

(continues on next page)



(continued from previous page)

```
CI_FILTER: ^NVCC|^GCC
CI_REORDER: ^GCC11 ^GCC8
```

The job generator looks for a line starting with the prefix `CI_FILTER` to filter the jobs or `CI_REORDER` to reorder the jobs. The filter statement is a single regex. The reorder statement can consist of multiple regex separated by a whitespace. For reordering, the jobs have the same order as the regex. This means that all orders matching the first regex are executed first, then the orders matching the second regex and so on. At the end, all orders that do not match any regex are executed. **Attention:** the order is only guaranteed across waves. Within a wave, it is not guaranteed which job will start first.

It is not necessary that both prefixes are used. One of them or none is also possible.

Hint: You can test your regex offline before creating and pushing a commit. The `job_generator.py` provides the `--filter` and `--reorder` flags that do the same thing as the lines starting with `CI_FILTER` and `CI_REORDER` in the commit message.

Hint: Each time the job generator runs it checks whether the container images exist. This is done by a request to the container registry which takes a lot of time. Therefore you can skip the check with the `--no-image-check` argument to speed up checking filters and reordering regex strings.

Develop new Feature for the alpaka-job-coverage Library

Sometimes one needs to implement a new function or fix a bug in the alpaka-job-coverage library while they are implementing a new function or fixing a bug in the alpaka job generator. Affected filter rules can be recognized by the fact that they only use parameters defined in this `globals.py`.

The following steps explain how to set up a development environment for the alpaka-job-coverage library and test your changes with the alpaka job generator.

We strongly recommend using a Python virtual environment.

```
# if not already done, clone repositories
git clone https://github.com/alpaka-group/alpaka-job-matrix-library.git
git clone https://github.com/alpaka-group/alpaka.git

cd alpaka-job-matrix-library
# link the files from the alpaka-job-matrix-library project folder into the site-
→packages folder of your environment
# make the package available in the Python interpreter via `import alpaka_job_
→coverage`
# if you change a src file in the folder, the changes are immediately available (if
→you use a Python interpreter instance, you have to restart it)
python setup.py develop
cd ..
cd alpaka
pip install -r script/job_generator/requirements.txt
```

Now you can simply run the alpaka job generator. If you change the source code in the project folder `alpaka-job-matrix-library`, it will be immediately available for the next generator run.

16.2.3 Custom jobs

You can create custom jobs that are defined as a yaml file. You can add the path of the folder to the function `add_custom_jobs()` in `script/job_generator/custom_job.py`. The function automatically read all files in the folder, which matches a filter function and loads the GitLab CI jobs. The custom jobs are added to the same job list as the generated jobs and distributed to the waves.

INDICES AND TABLES

- `genindex`
- `search`

INDEX

A

`alpaka::declareSharedVar` (C++ *function*), [58](#)
`ALPAKA_STATIC_ACC_MEM_CONSTANT` (C *macro*), [59](#)
`ALPAKA_STATIC_ACC_MEM_GLOBAL` (C *macro*), [59](#)