alpaka 0.5.0 Cheat Sheet

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`
- Color scheme: alpaka names, user-provided types and values
- This document assumes

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

Accelerator and Device

- Define in-kernel thread indexing type

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

- Define accelerator type (CUDA, OpenMP, etc.)

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

with AcceleratorType:

```
acc::AccGpuCudaRt,
acc::AccCpuOmp2Blocks,
acc::AccCpuOmp2Threads,
acc::AccCpuOmp4,
acc::AccCpuTbbBlocks,
acc::AccCpuThreads,
acc::AccCpuFibers,
acc::AccCpuSerial
```

- Select device for the given accelerator by index

```
auto const device =
pltf::getDevByIdx<Acc>(index);
```

Queue and Events

- Create a queue for a device

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

with Property: `queue::Blocking`, `queue::NonBlocking`

- Put a task for execution

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

- Wait for all operations in the queue

```
wait::wait(queue);
```

- Create an event

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

- Put an event to the queue

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

- Check if the event is completed

```
event::test(event);
```

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

```
wait::wait(event);
```

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 devHost = pltf::getDevByIdx<dev::DevCpu>(0u);
```

- Allocate a buffer in host memory

```
vec::Vec<Dim, Idx> extent = value;
using BufHost = mem::buf::Buf<DevHost, DataType, Dim, Idx>;
BufHost bufHost = mem::buf::alloc<DataType, Idx>(devHost, extent);
```

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

```
mem::buf::prepareForAsyncCopy(bufHost);
```

- Get a raw pointer to a buffer initialization, etc.

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

- Allocate a buffer in device memory

```
auto bufDevice = mem::buf::alloc<DataType, Idx>(device, extent);
```

- Enqueue a memory copy from host to device

```
mem::view::copy(queue, bufDevice, bufHost, extent);
```

- Enqueue a memory copy from device to host

```
mem::view::copy(queue, bufHost, bufDevice, extent);
```

Kernel Execution

- Automatically select a valid kernel launch configuration

```
vec::Vec<Dim, Idx> const globalThreadExtent = vectorValue;
vec::Vec<Dim, Idx> const elementsPerThread = vectorValue;
auto autoWorkDiv = workdiv::getValidWorkDiv<Acc>(device, globalThreadExtent,
    elementsPerThread, false,
    workdiv::GridBlockExtentSubDivRestrictions::Unrestricted);
```

- Manually set a kernel launch configuration

```
vec::Vec<Dim, Idx> const blocksPerGrid = vectorValue;
vec::Vec<Dim, Idx> const threadsPerBlock = vectorValue;
vec::Vec<Dim, Idx> const elementsPerThread = vectorValue;
using WorkDiv = 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 = kernel::createTaskKernel<Acc>(workDiv, kernel, parameters);
```

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

- Put the kernel for execution

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

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 = idx::getIdx<Origin, Unit>(acc);
auto extent = workdiv::getWorkdiv<Origin, Unit>(acc);
```

with `Origin`: `Grid`, `Block`, `Thread`

and `Unit`: `Blocks`, `Threads`, `Elems`

- Access components of multi-dimensional indices and extents

```
auto idxX = idx[0];
```

- Linearize multi-dimensional vectors

```
auto linearIdx = idx::mapIdx<1u>(idx, extent);
```

- Allocate static shared memory variable

```
Type & var = block::shared::st::allocVar<Type, __COUNTER__>(acc);
```

- Get dynamic shared memory pool, requires the kernel to specialize `kernel::traits::BlockSharedMemDynSizeBytes`

```
Type * dynamicSharedMemoryPool = block::shared::dyn::getMem<Type>(acc);
```

- Synchronize threads of the same block

```
block::sync::syncBlockThreads(acc);
```

- Atomic operations

```
auto result = atomic::atomicOp<Operation>(acc, arguments, OperationHierarchy);
```

with `Operation` (all in `atomic::op`): `atomic::op::Add`, `Sub`, `Min`, `Max`, `Exch`, `Inc`, `Dec`, `And`, `Or`, `Xor`, `Cas`

and `OperationHierarchy` (all in `hierarchy`): `hierarchy::Threads`, `Blocks`, `Grids`

- 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::generator::createDefault(acc, seed, subsequence);
auto number = distribution(generator);
```