Cheatsheet

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;
```

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);
```

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);
```

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

(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>;
// Create an alpaka vector which is a static array
alpaka::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);
```

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);
```

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

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<1u>(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);
```