

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 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,
AccCpuOmp2Blocks,
AccCpuOmp2Threads,
AccCpuOmp4,
AccCpuTbbBlocks,
AccCpuThreads,
AccCpuFibers,
AccCpuSerial
```

Select device for the given accelerator by index

```
auto const device = getDevByIdx<Acc>(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 devHost = getDevByIdx<DevCpu>(0u);
```

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);
```

Get a raw pointer to a buffer initialization, etc.

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

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, Elms

Access components of multi-dimensional indices and extents

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

Linearize multi-dimensional vectors

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

Allocate static shared memory variable

```
Type & var = declareSharedVar<Type, __COUNTER__>(acc);
```

Get dynamic shared memory pool, requires the kernel to specialize

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

Synchronize threads of the same block

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

Atomic operations

```
auto result = atomicOp<Operation>(acc,  
    arguments,  
    OperationHierarchy{});
```

Operation (all in op):

```
namespace op  
    Add, Sub, Min, Max, Exch, Inc, Dec, And, Or, Xor, Cas
```

OperationHierarchy (all in hierarchy):

```
namespace 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);
```