

# 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

```
// Use alpaka vector as a static array for the extents
Vec<DimInt<1>, Idx> extent = value;
Vec<DimInt<2>, Idx> extent = {valueY, valueX};

// Allocate memory for the alpaka buffer, which is a dynamic array
using BufHost = Buf<DevHost, DataType, Dim, Idx>;
BufHost bufHost = allocBuf<DataType, Idx>(devHost, extent);
```

### Create a view to host memory represented by a pointer

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

### Get the pitches of a buffer or view

```
// memory in bytes to the next element in the buffer/view along the pitch dimension
auto pitchBufOrViewAcc = getPitchesInBytes(accBufOrView)
```

### Get a mdspan to a buffer or view initialization, etc.

```
auto bufOrViewMdSpan = experimental::getMdSpan(bufOrViewAcc)
auto value = bufOrViewMdSpan(y,x); // access 2D mdspan
bufOrViewMdSpan(y,x) = value; // assign item to 2D mdspan
```

### Allocate a buffer in device memory

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

### Enqueue a memory copy from host to device

```
// arguments can be also View instances instead of Buf
memcpy(queue, bufDevice, bufHost, extent);
```

### Enqueue a memory copy from device to host

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

## Kernel Execution

### Prepare Kernel Bundle

```
HeatEquationKernel heatEqKernel;
```

### Automatically select a valid kernel launch configuration

```
Vec<Dim, Idx> const globalThreadExtent = vectorValue;  
Vec<Dim, Idx> const elementsPerThread = vectorValue;  
  
KernelCfg<Acc> const kernelCfg = {  
    globalThreadExtent,  
    elementsPerThread,  
    false,  
    GridBlockExtentSubDivRestrictions::Unrestricted};  
  
auto autoWorkDiv = getValidWorkDiv(  
    kernelCfg,  
    device,  
    kernel,  
    kernelParams...);
```

### 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 (does not launch it yet)

```
Kernel kernel{argumentsForConstructor};
```

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

### Get information about the kernel from the device (size, maxThreadsPerBlock, sharedMemSize, registers, etc.)

```
auto kernelFunctionAttributes = getFunctionAttributes<Acc>(devAcc, kernel, parameters...);
```

### Put the kernel for execution

```
exec(queue, workDiv, kernel, parameters...);
```

## 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<1u>(idxND, extentND);
```

### More generally, index multi-dimensional vectors with a different dimensionality

```
auto idxND = mapIdx<N>(idxMD, extentMD);
```

### Allocate static shared memory variable

```
Type& var = declareSharedVar<Type, COUNTER>(acc);  
auto& arr = declareSharedVar<float, 256, COUNTER>(acc); // scalar  
// 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);
```