# Abstracting reconstruction on accelerators: The Allen case in LHCb

Daniel Hugo Cámpora Pérez
Dorothea vom Bruch
dcampora@cern.ch

Compute Accelerator Forum, November 11th, 2020

Nikhef
Maastricht University
CNRS
CPPM

CERN

*LHCb*

Nikhef

Maastricht University
**Department of Data Science and
Knowledge Engineering**

## A heterogeneous computing framework

The Allen framework is a modular, scalable and flexible framework for physics reconstruction on accelerators.

Features:

- Supports CPU, CUDA, CUDACLANG and HIP targets. Possibly SYCL (tbd).
- Multi-threaded, pipelined, configurable framework.
- Multi-event scheduler (soon), event batches support.
- Custom memory manager, no dynamic allocations, flexible datatypes.
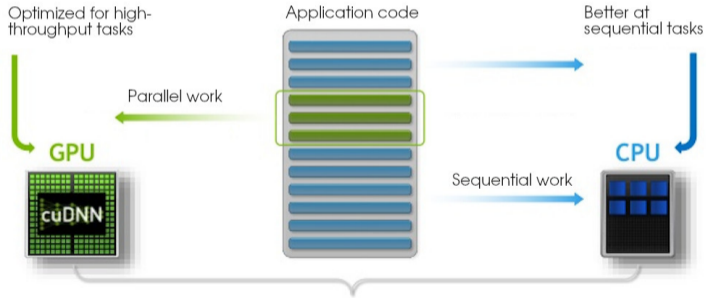- Built-in validation. Generation of graphs with ROOT.

## Host and device

## Host and device

The framework is geared towards using an accelerator to speed up parts of the computation. We can distinguish:

- Host processor – Processor that steers the computation.
- Device processor – Accelerator specialized for parallel processing.

## The device

The host is always a CPU. However, the device is configurable. Typically, the device is one of the following:

- A CPU.
- A GPU.
- An FPGA.

Allen supports several device as targets with the cmake option `TARGET_DEVICE`:

- `CPU` (default) – Sets the CPU as the device.
- `CUDA` – Targets an NVIDIA GPU, uses the `nvcc` compiler.
- `HIP` – Targets an AMD GPU with either `hipcc` or `clang-hip`.
- `CUDACLANG` – Targets an NVIDIA GPU with `clang`.

## Types of algorithms

Allen distinguishes two types of algorithms:

- **HostAlgorithm** – The execution of the algorithm happens *solely on the host.*
- **DeviceAlgorithm** – The algorithm offloads some of the work to the device.

## Kernels

In order to use the device, one has to define *kernels*. A **global kernel** (or *global function*) is a function that executes on the device.

**DeviceAlgorithms** use **global kernels** to offload execution to the device. Many kernels may be defined for a DeviceAlgorithm, although typically one is enough.

Eg. A *SAXPY* kernel:

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  for (unsigned i = threadIdx.x; i < n; i += blockDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

6

## Configuring kernel invocations

Generally when invoking a kernel, there are some configurable options, namely:

- `Grid dimension` – Number of blocks of kernel call.
- `Block dimension` – Number of threads per block.
- `Dynamic shared memory size` (not supported by Allen) – Size of dynamic shared memory.
- `Stream` – Stream that steers the execution.

Every kernel invocation can be configured differently. Usually a **property** is defined to be able to test the most efficient configuration in practice.

## Two processors, two memories

Host and device are two conceptually separate processors. Therefore, *host and device have two separate memories*.

In order to avoid *blocking memory allocation requests on the device*, we have created our own **memory manager**. Contrary to main memory, the device memory is limited in space and thus buffers are freed as soon as they are not needed anymore.

The memory manager can be configured as:

- Single allocation (default) – A single allocation happens on the memory manager instance of a user-defined memory amount. All subsequent malloc requests must fit under this memory amount, else they trigger a restart of the event batch with less events in the batch.
- Multiple allocations – Each malloc / free triggers a backend-specific call (eg. `memalign`, `cudaMalloc`, ...). Slower but allows to test out-of-bound accesses and better for memchecks.

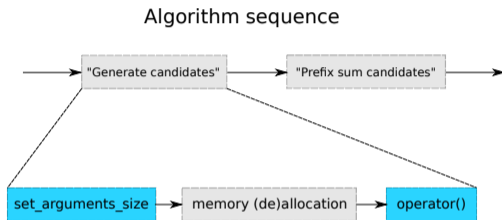We use an instance for the host and an instance for the device memory.
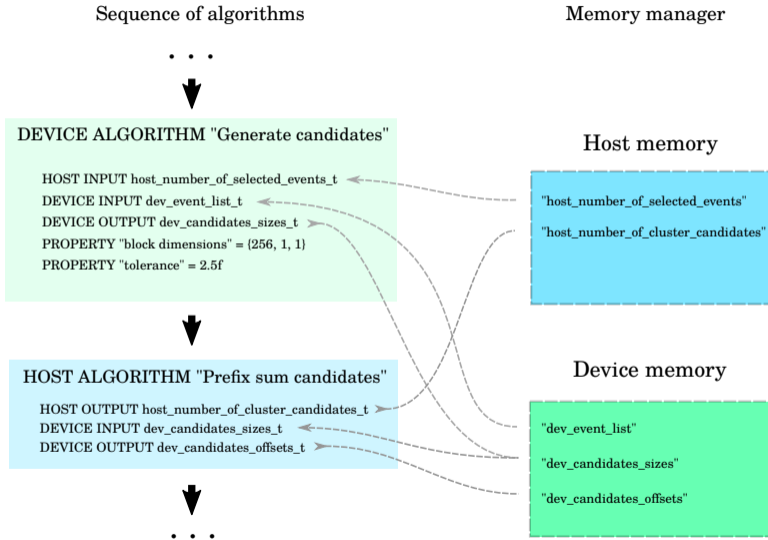
# Allen algorithms

Every algorithm must declare methods set_arguments_size and operator().

Algorithm sequence



If you are familiar with Gaudi / Athena, you may think of the operator() as in those frameworks. It is essentially where the algorithm action happens.

Allen requires an additional set_arguments_size method. **Explicit dynamic memory allocation (eg. std::vector) is not allowed in Allen,** and instead users must use this method to decide how much memory they needed.

## GenerateCandidates

As a brief example, we'll show a minimal implementation of DeviceAlgorithm *GenerateCandidates*.

- Header file GenerateCandidates.cuh.
- Source file GenerateCandidates.cu.

```
1  #pragma once
2  #include "DeviceAlgorithm.cuh"
3
4  namespace generate_candidates {
5    struct Parameters {
6      HOST_INPUT(host_number_of_selected_events_t, uint) host_number_of_selected_events;
7      DEVICE_INPUT(dev_event_list_t, uint) dev_event_list;
8      DEVICE_OUTPUT(dev_candidates_sizes_t, uint) dev_candidate_size;
9      PROPERTY(tolerance_t, "tolerance", "tolerance of search", float) tolerance;
10     PROPERTY(block_dim_t, "block_dim", "block dimensions", DeviceDimensions) block_dim;
11   };
12
13   struct generate_candidates_t : public DeviceAlgorithm, Parameters {
14     void set_arguments_size(
15       ArgumentReferences<Parameters>,
16       const RuntimeOptions&,
17       const Constants&,
18       const HostBuffers&) const;
19
20     void operator()(
21       const ArgumentReferences<Parameters>&,
22       const RuntimeOptions&,
23       const Constants&,
24       HostBuffers&,
25       const Allen::Context&) const;
26
27   private:
28     Property<tolerance_t> m_tolerance {this, 2.5f};
29     Property<block_dim_t> m_block_dim {this, {{256, 1, 1}}};
30   };
31 } // namespace generate_candidates
```

12

## Source GenerateCandidates.cu

```
1  #include "GenerateCandidates.cuh"
2
3  __global__ void generate_candidates::generate_candidates(generate_candidates::Parameters parameters)
4  {
5    // ...
6  }
7
8  void generate_candidates::generate_candidates_t::set_arguments_size(
9    ArgumentReferences<Parameters> arguments,
10   const RuntimeOptions&,
11   const Constants&,
12   const HostBuffers&) const
13  {
14   set_size<dev_candidates_size_t>(arguments, first<host_number_of_selected_events_t>(arguments));
15  }
16
17  void generate_candidates::generate_candidates_t::operator()(
18    const ArgumentReferences<Parameters>& arguments,
19    const RuntimeOptions&,
20    const Constants&,
21    HostBuffers&,
22    const Allen::Context& context) const
23  {
24   initialize<dev_candidates_sizes_t>(arguments, 0, context);
25   global_function(generate_candidates)(
26     dim3(first<host_number_of_selected_events_t>(arguments)),
27     property<block_dim_t>(),
28     context)(arguments);
29  }
```

## The CUDA kernel

Finally, we come to the global kernel definition. It accepts a single parameter:

generate_candidates::Parameters parameters

Now every input, output, and every property can be accessed by its **identifier**, and decays automatically to its **underlying type**.

```
1  // struct Parameters {
2  //   HOST_INPUT(host_number_of_selected_events_t, uint) host_number_of_selected_events;
3  //   DEVICE_INPUT(dev_event_list_t, uint) dev_event_list;
4  //   DEVICE_OUTPUT(dev_candidates_sizes_t, uint) dev_candidate_size;
5  //   PROPERTY(tolerance_t, "tolerance", "tolerance of search", float) tolerance;
6  //   PROPERTY(block_dim_t, "block_dim", "block dimensions", DeviceDimensions) block_dim;
7  // };
8
9  __global__ void generate_candidates::generate_candidates(generate_candidates::Parameters parameters)
10 {
11   const uint* a = parameters.dev_event_list;
12   uint* b = parameters.dev_candidate_size;
13   float c = parameters.tolerance;
14   DeviceDimensions d = parameters.block_dim;
15 }
```

# Backend

## Configurable targets

Allen can be configured with any of the following targets:

- CPU, CUDA, HIP, CUDACLANG, SYCL (*).

This choice is made at compile time through a cmake option. An Allen binary is therefore **compiled for a specific target device**.

## One source to rule them all

The Allen codebase is written once. The program generated depends on the configured *target device*.

- There are a few compromises to be made to achieve this. The maintainability and guarantee that the code is the same (and therefore likely to produce same results) are worth it.
- We have also developed a **thin** abstraction layer in the process, which is evolving.

## A distinction

- Utility functions – Functions used in the framework to carry out copy operations, explicit synchronizations, allocations, etc.
- Kernel code – Language used in kernel functions (ie. `__global__`, `__device__`, `__host__`).

## Utility functions language

A common wrapper which is specialized for each backend.

```cpp
namespace Allen {
  // Holds an execution context. An execution
  // context allows to execute kernels in parallel,
  // and provides a manner for execution to be stopped.
  struct Context;

  // Utility functions
  void malloc(void** devPtr, size_t size);
  void malloc_host(void** ptr, size_t size);
  void memcpy(void* dst, const void* src, size_t count, enum memcpy_kind kind);
  void memcpy_async(void* dst, const void* src, size_t count, enum memcpy_kind kind, const Context& context);
  void memset(void* devPtr, int value, size_t count);
  void memset_async(void* ptr, int value, size_t count, const Context& context);
  void free_host(void* ptr);
  void free(void* ptr);
  void synchronize(const Context& context);
  void device_reset();
  void peek_at_last_error();
  void host_unregister(void* ptr);
  void host_register(void* ptr, size_t size, enum host_register_kind flags);
} // namespace Allen
```

## Utility functions examples

```
1  // CPU implementations
2  void inline malloc(void** devPtr, size_t size) { posix_memalign(devPtr, 64, size); }
3  void inline memcpy(void* dst, const void* src, size_t count, Allen::memcpy_kind)
4  {
5    std::memcpy(dst, src, count);
6  }
7
8  // CUDA / HIP implementations
9  void inline malloc(void** devPtr, size_t size) { cudaCheck(cudaMalloc(devPtr, size)); }
10 void inline memcpy(void* dst, const void* src, size_t count, Allen::memcpy_kind kind)
11 {
12   cudaCheck(cudaMemcpy(dst, src, count, convert_allen_to_cuda_kind(kind)));
13 }
14
15 // SYCL implementations
16 void inline malloc(void** ptr, size_t size)
17 {
18   *ptr = sycl::malloc_device(size, global_queue);
19   global_queue.wait_and_throw();
20 }
21 void inline memcpy(void* dst, const void* src, size_t count, Allen::memcpy_kind)
22 {
23   global_queue.memcpy(dst, src, count);
24   global_queue.wait_and_throw();
25 }
```

## Kernel code language

We use CUDA in all Allen kernel code. There are several reasons to this choice:

- CUDA is by far the most widely used language for GPU acceleration.
- There are many resources for (not-so-)new developers, and developers are more likely to succeed.
- This is where you will most likely spend a lot of time optimizing. Performance is a top priority.
- CUDA is pleasantly evolving to support the latest features of the C++ standard.

## Kernel code language

We use CUDA in all Allen kernel code. There are several reasons to this choice:

- CUDA is by far the most widely used language for GPU acceleration.
- There are many resources for (not-so-)new developers, and developers are more likely to succeed.
- This is where you will most likely spend a lot of time optimizing. Performance is a top priority.
- CUDA is pleasantly evolving to support the latest features of the C++ standard.

Note: Allen is not aiming at supporting every CUDA functionality. Typically, few constructs suffice (ie. `__syncthreads()`, `atomic operations`, `thread / block indices`, `grid / block dimensions`). If a developer intends to use a low-level construct that does not exist, they should provide compatibility code.

## Behind the scenes: HIP backend

AMD is developing the HIP language to be as close as possible to CUDA. Hence, the compatibility between CUDA and HIP is rather simple by design, with very few exceptions.

```
1  // CUDA to HIP conversion
2  #define cudaMalloc hipMalloc
3  #define cudaMallocHost hipHostMalloc
4  #define cudaMemcpy hipMemcpy
5  #define cudaMemcpyAsync hipMemcpyAsync
6  ...
```

HIP is meant as a language that can compile and run in either NVIDIA or AMD architectures. However, since performance is a top priority, we in Allen want to support the **native code generation** of each vendor.

Consider the following CUDA code:

```
1  constexpr int N = 32;
2  __global__ void saxpy_plus(float* x, float* y, const float a) {
3    y[threadIdx.x] = x[threadIdx.x] * a + y[threadIdx.x];
4    __syncthreads();
5    if (threadIdx.x < 10) {
6      y[i] += 1;
7    }
8    if (threadIdx.x == 10) {
9      y[threadIdx.x] += 20;
10   }
11 }
12 ...
13 saxpy_plus<<</*blocks*/ M, /*threads*/ N>>>(x, y, a);
```

- The number of threads is set statically to N=32.
- The statement in line 3 makes assumptions of the number of threads.
- The two if statements also make assumptions of the number of threads (they require at least 11 threads).

## Flexible code

In contrast, consider this code:

```
1  constexpr int N = 32;
2  __global__ void saxpy_plus(float* x, float* y, const float a) {
3    for (int i=threadIdx.x; i<N; i+=blockDim.x) {
4      y[i] = x[i] * a + y[i];
5    }
6    __syncthreads();
7    for (int i=threadIdx.x; i<10; i+=blockDim.x) {
8      y[i] += 1;
9    }
10   if (threadIdx.x == 0) {
11     y[10] += 20;
12   }
13 }
14 ...
15 saxpy_plus<<</*blocks*/ M, /*threads*/ 1>>>(x, y, a);
```

- A call to saxpy_plus with any number of threads will produce the same result.

## A CPU version

If the CUDA code satisfies that it produces the same result when invoked with a block dimension of
$\{1, 1, 1\}$ – or in other words:

- `for`-loops over threads are *block-dimension strided*.
- `if`-statements for a single thread refer to threads of index 0.

Then, with some macros and function definitions it is possible to compile the code for CPUs.

```
1  // Definitions excerpt
2  thread_local GridDimensions gridDim;
3  thread_local BlockIndices blockIdx;
4  constexpr BlockDimensions blockDim {1, 1, 1};
5  constexpr ThreadIndices threadIdx {0, 0, 0};
6  ...
7  // Kernel call excerpt
8  gridDim = {num_blocks.x, num_blocks.y, num_blocks.z};
9  for (unsigned int i = 0; i < num_blocks.x; ++i) {
10   for (unsigned int j = 0; j < num_blocks.y; ++j) {
11     for (unsigned int k = 0; k < num_blocks.z; ++k) {
12       blockIdx = {i, j, k};
13       function(std::get<I>(invoke_arguments)...);
14     }
15   }
16 }
```

## Multi-threading

The previously shown example is single-threaded. There are several manners to achieve multi-threading:

1. Make every kernel call a multi-threaded call. Each thread would have their own `threadIdx`, and `__syncthreads()` would have to be a barrier that synchronizes threads.
2. Execute blocks in separate threads. In CUDA code block synchronization is very rare and expensive, so in practice each thread would be able to execute its own code and join at the end of the kernel execution.
3. Execute CUDA streams in separate threads.

Since Allen already supports running multiple concurrent sequences in parallel, we opted for option 3 (which scales best).

## CPU backend

In short, Allen for CPU is:

- multi-threaded.
- supported across architectures (tested x86_64, ARM, PowerPC).
- a compilation of the Allen codebase. No extra maintenance.

# SYCL support in Allen (brief)

## The evolving SYCL standard

SYCL is a rapidly evolving standard that Intel is steering. It builds on top of the foundations OpenCL set and incorporates new C++ good practices coming from the latest C++ standards.

SYCL supports a variety of architectures from the get-go:

- CPU
- Intel GPUs
- Intel FPGAs
- NVIDIA GPUs
- AMD GPUs

Recently, SYCL 2020 was released, with more functionality that makes it easier to port already existing codebases to it.

The recent SYCL changes have allowed us to develop a prototype of Allen's VELO reconstruction. We expect the prototype to be extensible to other parts of Allen.

In order to do so, we had to alter part of the backbone of Allen to support SYCL-specific requirements. More details of our ongoing MR (`https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/443`).

Some of those requirements are rather... confusing to say the least.

## Our experience

Supporting utility functions was not a problem.

The bigger issue however is that **supporting SYCL would force us to move away from CUDA as a kernel language**. Even though it is possible, this has deep implications:

- SYCL is intended as a one-solution-fits-all. Developers are inteded to abandon CUDA and adopt SYCL with the promise that compiling SYCL will retain the performance.
- However, we want to continue to support **native code generation**.
- Therefore, if we want to support SYCL we would have to write our own middle-language for kernels, at which point SYCL becomes more of a hassle rather than a help.
- The inexistence of SYCL examples online and the status of the documentation prove as a hard entry point.

## Example: What it means for Allen

```
1  // Before
2  __device__ foo() {
3    threadIdx.x;
4    blockIdx.z;
5    __syncthreads();
6  }
7
8  __global__ some_kernel(Parameters parameters) {
9    foo();
10 }
```

```
1  // After
2  __device__ foo(const Allen::KernelInvocationConfiguration& config) {
3    config.threadIdx<0>();
4    config.blockIdx<2>();
5    config.syncthreads();
6  }
7
8  __global__ some_kernel(Parameters parameters) {
9    const auto& config = parameters.config;
10   foo(config);
11 }
```

*We are still evaluating the consequences of adding SYCL support in Allen.*

# A word about configuration (brief)

## A sequence of algorithms

Allen centers around the idea of running a *sequence of algorithms* on input events. This sequence is predefined and will always be executed in the same order.

- The sequence can be configured with python.
- Existing configurations can be browsed under `configuration/sequences`.
- A configuration name is the name of each individual file, without the `.py` extension
- Sequences are chosen *at compile time* with cmake option `SEQUENCE`.

Eg. `cmake -DSEQUENCE=velo ..`

## Inspecting algorithms (1)

All the code is parsed with a *libClang* parser when the cmake command is executed.

It is possible (and encouraged) to inspect the parsed definitions of algorithms.

```
 1  cmake ..
 2  foo@bar:build$ cmake ..
 3  foo@bar:build$ cd sequences
 4  foo@bar:build/sequences$ python3
 5  Python 3.8.2 (default, Feb 28 2020, 00:00:00)
 6  [GCC 10.0.1 20200216 (Red Hat 10.0.1-0.8)] on linux
 7  Type "help", "copyright", "credits" or "license" for more information.
 8  >>> from definitions import algorithms
 9  >>> algorithms.velo_
10  algorithms.velo_calculate_number_of_candidates_t(   algorithms.velo_kalman_filter_t(
11  algorithms.velo_calculate_phi_and_sort_t(           algorithms.velo_masked_clustering_t(
12  algorithms.velo_consolidate_tracks_t(               algorithms.velo_pv_ip_t(
13  algorithms.velo_copy_track_hit_number_t(            algorithms.velo_search_by_triplet_t(
14  algorithms.velo_estimate_input_size_t(              algorithms.velo_three_hit_tracks_filter_t(
```

32

## Inspecting algorithms (2)

One can see the input and output parameters and properties of an algorithm by just printing the class representation of an algorithm (ie. *without parentheses*).

```
1  >>> algorithms.velo_calculate_number_of_candidates_t
2  class AlgorithmRepr : DeviceAlgorithm
3   inputs: ('host_number_of_selected_events_t', 'dev_event_list_t', 'dev_velo_raw_input_t', 'dev_velo_raw_input_offsets_t')
4   outputs: ('dev_number_of_candidates_t',)
5   properties: ('block_dim_x',)
```

## Multi-event scheduler

There is an ongoing Merge Request in Allen (`https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/429`) with a multi-event scheduler. It is rather complex, and probably a topic for another meeting.

Allen will become configurable *à la Gaudi* with that change, adopting the same convention and a very similar frontend, familiar to users.

If you are interested, drop us an e-mail or follow the above MR thread.

# Conclusions

## An evolving design

- Allen allows developers to write algorithms, kernels, and to configure them.
- We are hiding the complexity away and giving a common look-and-feel.
- Low entry-level, maintainability and performance are top priorities.

- Our current abstraction has been very successful considering Allen's short development cycle.
- Going forward, we want to extend on these features supporting relevant hardware.
- Our abstraction will likely grow with time (eg. ML frameworks, other libraries).
- We intend to establish solid foundations and grow organically.