Reconstruction on accelerators with Allen

Daniel Hugo Cámpora Pérez
dcampora@cern.ch

ACTS parallelization meeting, March 19th, 2021

Maastricht University

A heterogeneous computing framework

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

Features:

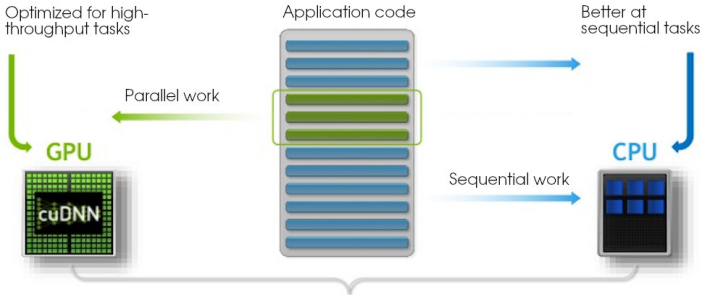
- Supports CPU, CUDA and HIP targets.
- Uses the latest common available C++ standard (C++17).
- Multi-threaded, pipelined, configurable framework.
- Multi-event scheduler (soon), event batches support.
- Custom memory manager, no dynamic allocations, flexible datatypes.
- Built-in validation. Generation 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 `hipcc`.
- `CUDACLANG` (best-effort) – Targets an NVIDIA GPU with `clang`.

Types of algorithms

Allen distinguishes several types of algorithms. Algorithms are not restricted in any way, and the type of algorithm is only picked up by the code generation steps of Allen:

- **HostAlgorithm** – The execution of the algorithm happens *solely on the host*.
- **DeviceAlgorithm** – The algorithm offloads some of the work to the device.
- **SelectionAlgorithm** – An algorithm that performs a selection.
- **ValidationAlgorithm** – An algorithm that performs a validation. The presence of one ValidationAlgorithm in the sequence indicates that the sequence requires MC data.

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];
  }
}
```

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.

Execution of Allen

The execution of Allen's sequence loop is steered by a single program which starts a configurable number of threads.

- Each thread instantiates a stream. We refer to these as a thread / stream pair.
- Each thread / stream runs subsequent full sequences of algorithms, independently from the other threads / streams.
- Each thread / stream never instantiates new streams or new threads.

A thread / streams executes on the resources it logically has access to from its thread / stream. A thread means a single thread, a stream means possibly the entire device.

Note: Separately from the sequence of algorithms, Allen also performs DAQ-related tasks such as fetching events in slices, shuffling them, handing them to the sequence, monitoring, etc. which are not covered in this presentation.

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.

The best configuration

Allen can either run in ”benchmark mode” or fetch a continuous stream of data. In benchmark mode, the application iterates over the same events in round-robin.

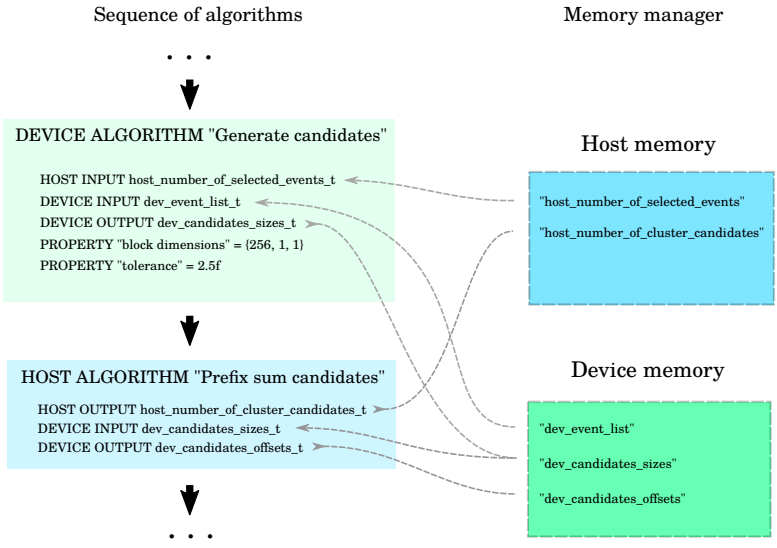
Parameters that impact performance:

- `-t` – Number of thread/streams.
- `-n X --events-per-slice X` – Number of events in a batch of events.
- `-m` – Memory for each thread/stream.
- `-r` – Number of repetitions.

For instance, the current configuration we use in LHCb to test performance is `-t 16 -n 500 --events-per-slice 500 -m 500 -r 1000`. This translates into $16 \cdot 500 = 8GB$ of memory utilization for the sequences, and $16 \cdot 500 \cdot 1000 = 8M$ events run.

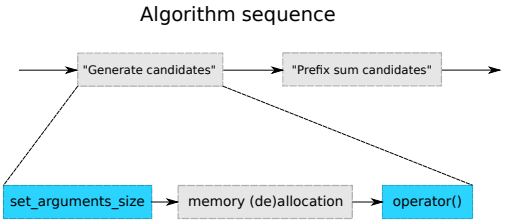
Allen algorithms

Types of algorithms



Declaration of two methods

Every algorithm must declare methods `set_arguments_size` and `operator()`.



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`.

Header GenerateCandidates.cuh

```
#pragma once
#include "DeviceAlgorithm.cuh"

namespace generate_candidates {
  struct Parameters {
    HOST_INPUT(host_number_of_selected_events_t, uint) host_number_of_selected_events;
    DEVICE_INPUT(dev_event_list_t, uint) dev_event_list;
    DEVICE_OUTPUT(dev_candidates_sizes_t, uint) dev_candidate_size;
    PROPERTY(tolerance_t, "tolerance", "tolerance␣of␣search", float) tolerance;
    PROPERTY(block_dim_t, "block_dim", "block␣dimensions", DeviceDimensions) block_dim;
  };

  struct generate_candidates_t : public DeviceAlgorithm, Parameters {
    void set_arguments_size(
      ArgumentReferences<Parameters>,
      const RuntimeOptions&,
      const Constants&,
      const HostBuffers&) const;

    void operator()(
      const ArgumentReferences<Parameters>&,
      const RuntimeOptions&,
      const Constants&,
      HostBuffers&,
      const Allen::Context&) const;

  private:
    Property<tolerance_t> m_tolerance {this, 2.5f};
    Property<block_dim_t> m_block_dim {this, {{256, 1, 1}}};
  };
} // namespace generate_candidates
```

Source GenerateCandidates.cu

```
#include "GenerateCandidates.cuh"

__global__ void generate_candidates::generate_candidates(generate_candidates::Parameters parameters)
{
  // ...
}

void generate_candidates::generate_candidates_t::set_arguments_size(
  ArgumentReferences<Parameters> arguments,
  const RuntimeOptions&,
  const Constants&,
  const HostBuffers&) const
{
  set_size<dev_candidates_size_t>(arguments, first<host_number_of_selected_events_t>(arguments));
}

void generate_candidates::generate_candidates_t::operator()(
  const ArgumentReferences<Parameters>& arguments,
  const RuntimeOptions&,
  const Constants&,
  HostBuffers&,
  const Allen::Context& context) const
{
  initialize<dev_candidates_sizes_t>(arguments, 0, context);
  global_function(generate_candidates)(
    dim3(first<host_number_of_selected_events_t>(arguments)),
    property<block_dim_t>(),
    context)(arguments);
}
```

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**.

```
// struct Parameters {
//   HOST_INPUT(host_number_of_selected_events_t, uint) host_number_of_selected_events;
//   DEVICE_INPUT(dev_event_list_t, uint) dev_event_list;
//   DEVICE_OUTPUT(dev_candidates_sizes_t, uint) dev_candidate_size;
//   PROPERTY(tolerance_t, "tolerance", "tolerance of search", float) tolerance;
//   PROPERTY(block_dim_t, "block_dim", "block dimensions", DeviceDimensions) block_dim;
// };

__global__ void generate_candidates::generate_candidates(generate_candidates::Parameters parameters)
{
  const uint* a = parameters.dev_event_list;
  uint* b = parameters.dev_candidate_size;
  float c = parameters.tolerance;
  DeviceDimensions d = parameters.block_dim;
}
```

Backend

Configurable targets

Allen can be configured with any of the following targets:

- CPU, CUDA, HIP, CUDACLANG (best-effort).

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.

```
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

```
// CPU implementations
void inline malloc(void** devPtr, size_t size) { posix_memalign(devPtr, 64, size); }
void inline memcpy(void* dst, const void* src, size_t count, Allen::memcpy_kind)
{
  std::memcpy(dst, src, count);
}

// CUDA implementations
void inline malloc(void** devPtr, size_t size) { cudaCheck(cudaMalloc(devPtr, size)); }
void inline memcpy(void* dst, const void* src, size_t count, Allen::memcpy_kind kind)
{
  cudaCheck(cudaMemcpy(dst, src, count, convert_allen_to_cuda_kind(kind)));
}

// HIP implementations
void inline malloc(void** devPtr, size_t size) { hipCheck(hipMalloc(devPtr, size)); }
void inline memcpy(void* dst, const void* src, size_t count, Allen::memcpy_kind kind)
{
  hipCheck(hipMemcpy(dst, src, count, convert_allen_to_hip_kind(kind)));
}
```

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: backends

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.

- In fact, the kernel language is the same and it has taken no effort so far to maintain it.
- In order to account for the utils minor differences, a separate backend is included depending on the target detected at compile time.

```
|-- include
|   |-- BackendCommon.h
|   |-- BackendCommonInterface.h
|   |-- CPUBackend.h
|   |-- CUDABackend.h
|   |-- HIPBackend.h
...
```

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.

Behind the scenes: CPU backend

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:

```
constexpr int N = 32;
__global__ void saxpy_plus(float* x, float* y, const float a) {
  for (int i=threadIdx.x; i<N; i+=blockDim.x) {
    y[i] = x[i] * a + y[i];
  }
  __syncthreads();
  for (int i=threadIdx.x; i<10; i+=blockDim.x) {
    y[i] += 1;
  }
  if (threadIdx.x == 0) {
    y[10] += 20;
  }
}
...
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.

```
// Definitions excerpt
thread_local GridDimensions gridDim;
thread_local BlockIndices blockIdx;
constexpr BlockDimensions blockDim {1, 1, 1};
constexpr ThreadIndices threadIdx {0, 0, 0};
...
// Kernel call excerpt
gridDim = {num_blocks.x, num_blocks.y, num_blocks.z};
for (unsigned int i = 0; i < num_blocks.x; ++i) {
  for (unsigned int j = 0; j < num_blocks.y; ++j) {
    for (unsigned int k = 0; k < num_blocks.z; ++k) {
      blockIdx = {i, j, k};
      function(std::get<I>(invoke_arguments)...);
    }
  }
}
```

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.

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.

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

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*).

```
>>> algorithms.velo_calculate_number_of_candidates_t
class AlgorithmRepr : DeviceAlgorithm
 inputs: ('host_number_of_selected_events_t', 'dev_event_list_t', 'dev_velo_raw_input_t', 'dev_velo_raw_input_offsets_t')
 outputs: ('dev_number_of_candidates_t',)
 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.

Thanks for your attention!