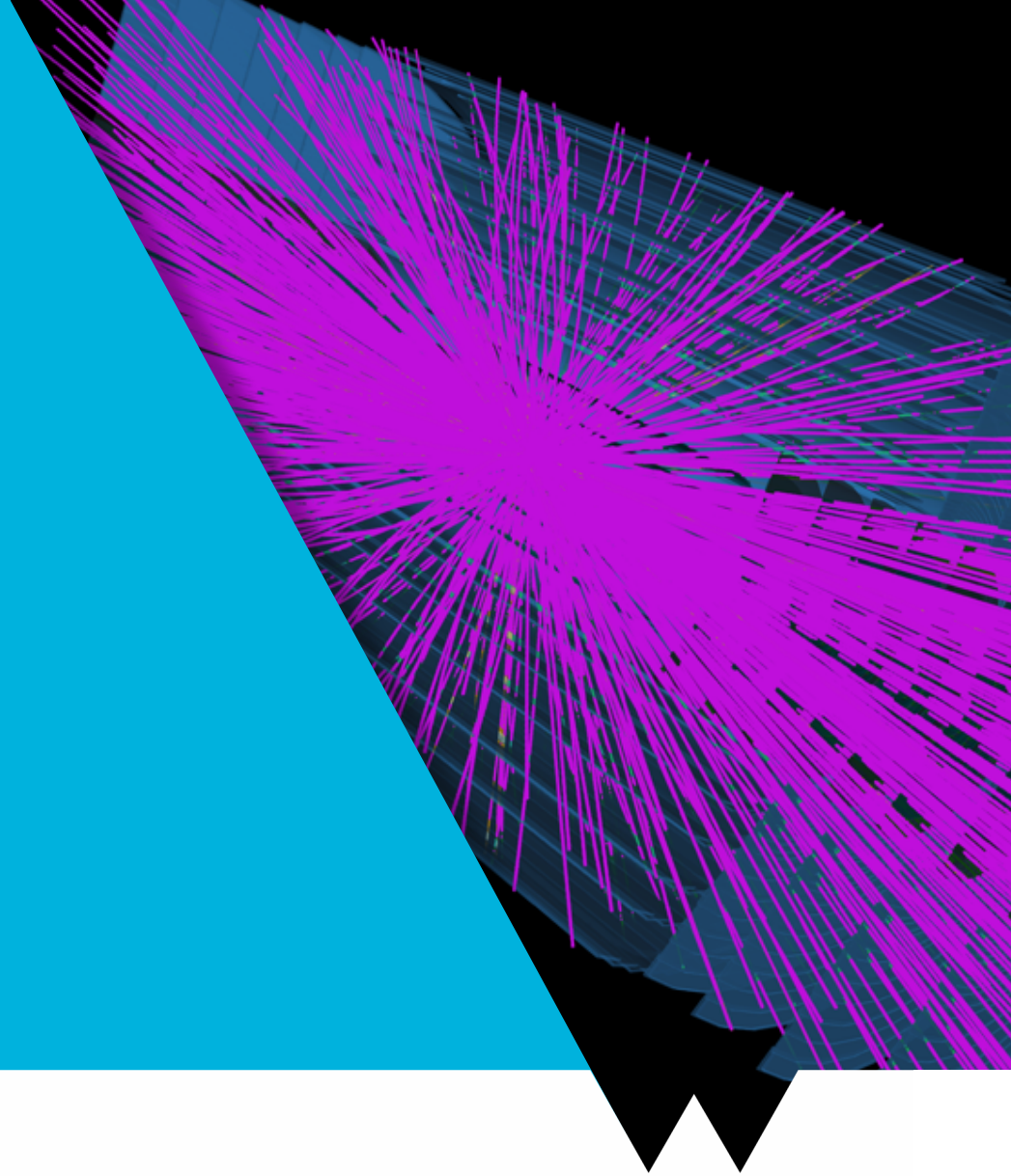
traccc & alpaka

Thoughts & First Look

Ryan Cross - GridPP and Swift-HEP

2023/03/30



WARWICK
THE UNIVERSITY OF WARWICK

Overview

This talk will cover:

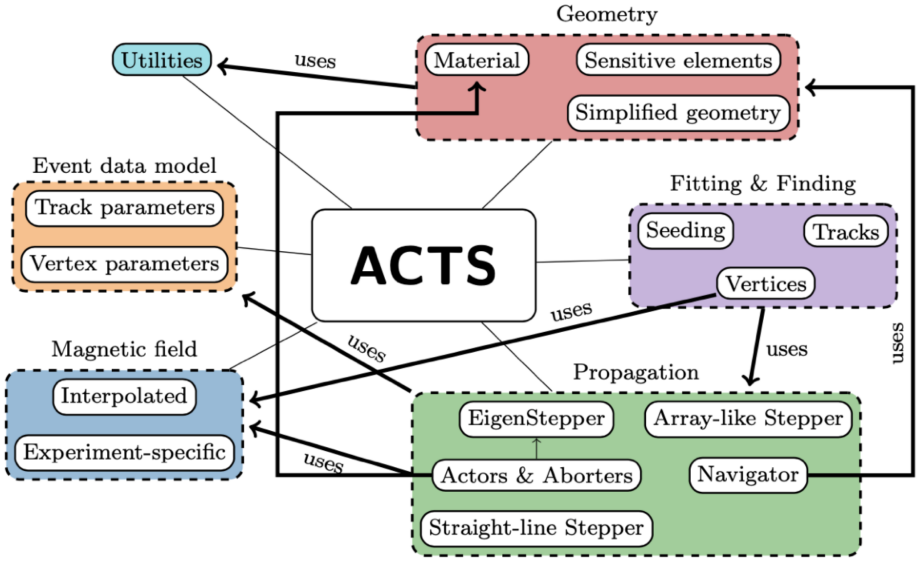
1. **A quick overview of traccc.**
2. **Cross-Platform Abstraction Libraries.**
3. **First Steps with alpaka.**
4. **What comes next?**

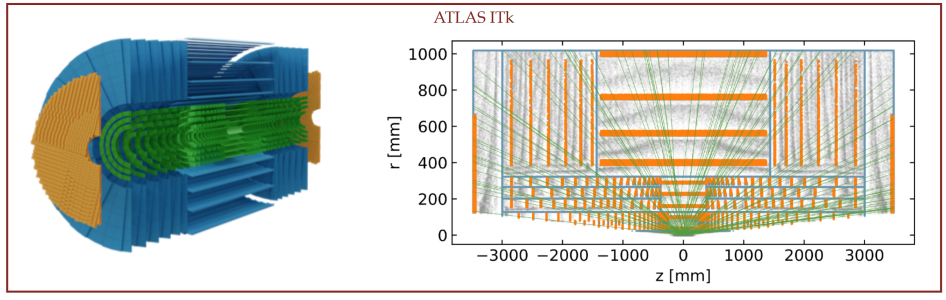
A Common Tracking Software

ACTS is a generic, experiment independent framework/software toolkit, written in C++. Through it, you can get algorithms for track reconstruction that can be used in any experiment, agnostic of any technical details (detector tech, design and event processing framework).

It has been designed in a thread-safe manner, with support for parallel code execution and optimised data structures for speeding up the many linear algebra operations used throughout the code base.

Wide set of use cases, with integrations/progress for Belle II, CEPC, sPHENIX, PANDA, FASER, ATLAS ID (current + ITk).



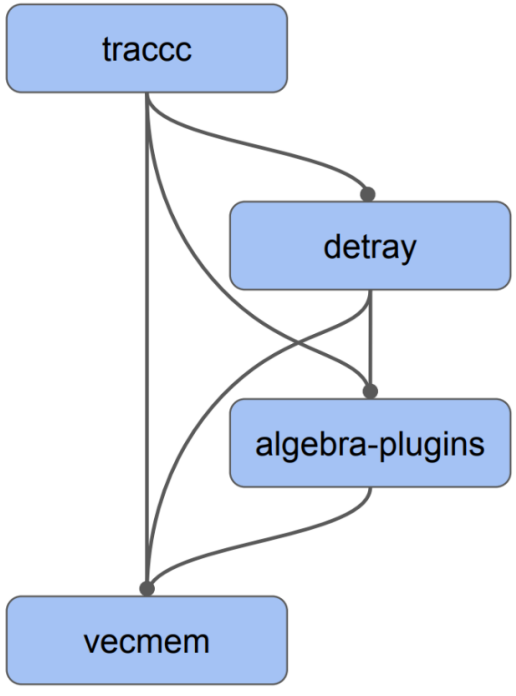


ACTS R&D Projects

Many of the core algorithms in ACTS have been ported to CUDA and SYCL, but there is a limit as to how far this can go. Full offloading is difficult, with some of the event data model and geometry not being the most GPU-friendly.

To tackle this, ACTS has launched several R&D projects:

- traccc - Tracking Algorithms on the GPU.
- detray - A GPU based Geometry Builder.
- algebra-plugin - Provides varying algebra plugins for the other projects.
- vecmem - A GPU Memory Management Tool for the other projects.

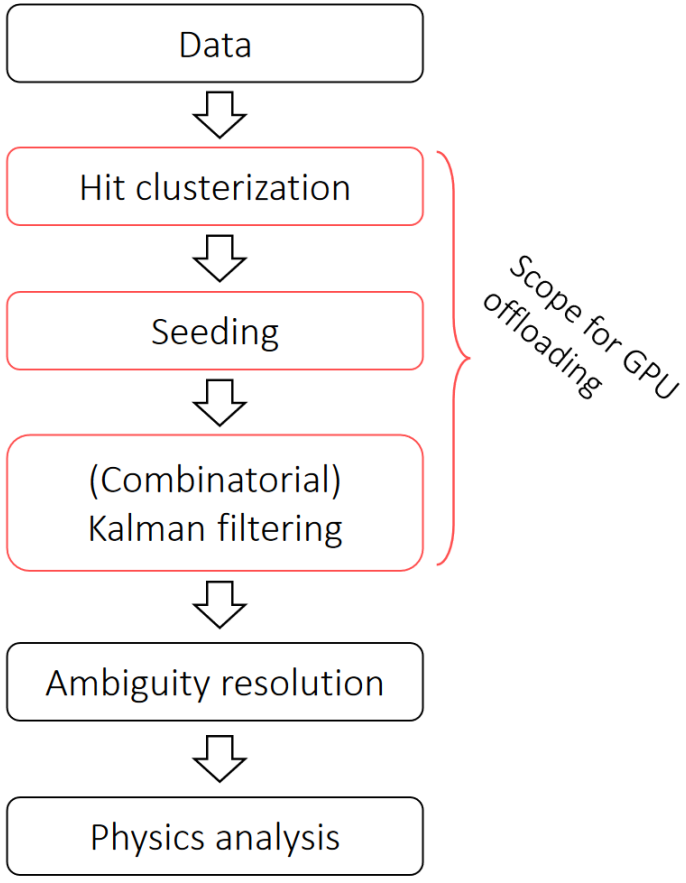


traccc

traccc, as mentioned, is being developed to find an optimal solution for utilising GPUs in ACTS. This means:

- Establish an **event data model** (EDM) that is interoperable between both the CPU and GPU.
  - This utilises the vecmem project for memory management, which supports a number of backends.
  - Allows the user to choose an architecture just by utilising a specific memory resource.

- The algorithms are setup in such a way to **exploit parallelisation architecture** available on GPUs.
  - Avoid dynamic allocations and branching.
  - Fully distribute tasks against the GPU.



Cross-Platform Abstraction - Why?

As part of traccc, the use of abstraction tools are interesting. There are many different ways to write code that can run on a GPU.

Writing CUDA directly can be a mixed experience. You are locked into a specific vendor for acceleration, but you don't have to deal with the additional layer of complexity an abstraction library brings.

Because of this, there exists many different abstraction paradigms, software that allows a single source code base that at compile time can target many architectures. Common targets include CUDA, AMD / Intel GPUs, and CPU parallelism via `thread`, OpenMP, TBB and more.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

- **SYCL** is a higher level programming model, developed by the Khronos group (OpenCL/OpenGL/Vulkan and more). It defines an abstraction layer that enables code for heterogeneous processors via a 'single-source' style in standard C++. Supports many backends: CUDA, AMD GPUs, Intel GPUs, OpenMP, MPI, Vulkan, `std::thread`, OpenCL and more.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

- **SYCL** is a higher level programming model, developed by the Khronos group (OpenCL/OpenGL/Vulkan and more). It defines an abstraction layer that enables code for heterogeneous processors via a 'single-source' style in standard C++. Supports many backends: CUDA, AMD GPUs, Intel GPUs, OpenMP, MPI, Vulkan, `std::thread`, OpenCL and more.

- **Kokkos** is C++ based programming model, which provides methods that abstract away details of parallel execution and memory management, such that code can be written for many shared-memory programming models in a unifed way. Supports CUDA, HIP, SYCL, HPX, OpenMP and `std::thread`.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

- **SYCL** is a higher level programming model, developed by the Khronos group (OpenCL/OpenGL/Vulkan and more). It defines an abstraction layer that enables code for heterogeneous processors via a 'single-source' style in standard C++. Supports many backends: CUDA, AMD GPUs, Intel GPUs, OpenMP, MPI, Vulkan, `std::thread`, OpenCL and more.

- **Kokkos** is C++ based programming model, which provides methods that abstract away details of parallel execution and memory management, such that code can be written for many shared-memory programming models in a unifed way. Supports CUDA, HIP, SYCL, HPX, OpenMP and `std::tread`.

- **alpaka** is a header-only C++ 17 abstraction library for accelerator development. It aims to provide performance portability acorss a range of accelerators through the abstraction of the underlying levels of parallelism. Support CUDA, OpenMP, `std::thread`, TBB, HIP and OpenAcc.

kokkos

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Get an accelerator device:

```
accelerator = getAcceleratorDevice();
queue = getDeviceQueue(accelerator);
```

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Get an accelerator device:

```
accelerator = getAcceleratorDevice();
queue = getDeviceQueue(accelerator);
```

Define an operation for the device to perform:

```
job = [](auto accelerator, auto config, auto items) {
    auto item = items[getThreadIndex()];
    ...
};
```

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Get an accelerator device:

```
accelerator = getAcceleratorDevice();
queue = getDeviceQueue(accelerator);
```

Define an operation for the device to perform:

```
job = [](auto accelerator, auto config, auto items) {
    auto item = items[getThreadIndex()];
    ...
};
```

Run the jobs in parallel:

```
queue.submit(job, configuration, items);
queue.wait();
```

Why alpaka?

I've just outlined three projects that support the "write once, support many" paradigm, and both SYCL and Kokkos are already implemented in traccc, with differing levels of functionality. So why a third?

alpaka was chosen as a possible candidate for a few reasons:

- **Simplicity**: alpaka is a lightweight, header-only library, which makes integration into traccc very easy, as well as it being written in the same modern C++17 as traccc/acts.
- **Familiarity**: The alpaka abstraction model is very similar to the CUDA grid-blocks-thread model, making writing code for alpaka simple, and familiar for those with CUDA experience, whilst also providing a CPU and non-CUDA based implementation.
- **Community Support**: alpaka has been used extensively at CMS, including in `cms-sw` and their HLT achieving performance close to that of the native CUDA codebase, from a single source code that can be utilised on many devices.

First Steps

The first steps around integration of alpaka in traccc were performed by Stewart Martin-Haugh, as part of a PR in January: PR #300.

That implemented all the basic building blocks for alpaka inside of traccc:

- Building of alpaka (or using a system install) for integration.
- A basic test of alpaka to setup an accelerator and then perform a dummy calculation on it, inside the test infrastructure of traccc.

That gives a solid base on which to iterate, now that the more abstract bits are finished and alpaka is available to utilise in modules.

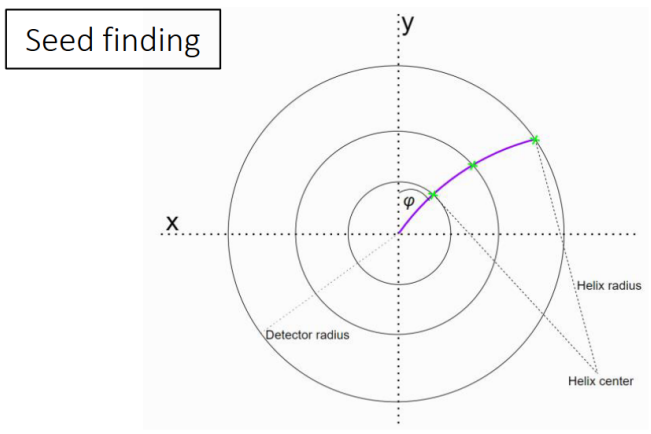
```
struct VectorOpKernel {
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const& acc, float* result, uint32_t n) const {

        auto const threadIdx = getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u];
        if (threadIdx < n)
            result[threadIdx] = alpaka::math::sin(acc, threadIdx);
    }
};
```

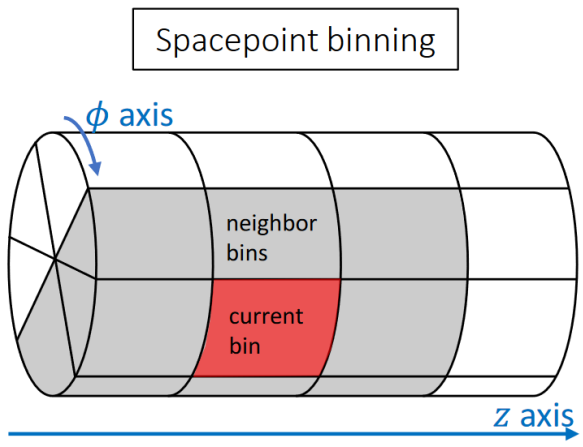
Implementation of First Algorithm - Process

Prior to alpaka, Kokkos is the most recently added backend to traccc, so the first step was to work on the module that kokkos had implemented first, which is the spacepoint binning step, which forms part of the seeding example.

Initially, I started by using the Kokkos code as a basic reference to understand the code layout and approach, but it quickly became clear that alpaka is much closer to a native CUDA implementation, rather than the approach Kokkos takes.

Broadly, the CUDA implementation can be converted to an alpaka one by wrapping the kernels used into alpaka `ALPAKA_FN_ACC` kernels, and utilising the helper wrappers to access the thread/block/grid index, rather than the CUDA-supplied interface.





Implementation of First Algorithm - Example

To show some examples, here is some comparisons between the CUDA and alpaka implementation for the this first spacepoint binning step.

Initialising the kernel running parameters

```
const unsigned int num_threads = WARP_SIZE * 8;
const unsigned int num_blocks = (sp_size + num_threads - 1) / num_threads;
```

```
auto const deviceProperties = alpaka::getAccDevProps<Acc>(devAcc);
auto const maxThreadsPerBlock = deviceProperties.m_blockThreadExtentMax[0];
auto const threadsPerBlock = maxThreadsPerBlock;
auto const blocksPerGrid = (sp_size + threadsPerBlock - 1) / threadsPerBlock;
auto const elementsPerThread = 1u;
```

Implementation of First Algorithm - Example

To show some examples, here is some comparisons between the CUDA and alpaka implementation for the this first spacepoint binning step.

Defining a spacepoint-based kernel

```
__global__ void populate_grid(seedfinder_config config,
    spacepoint_collection_types::const_view spacepoints, sp_grid_view grid) {
    device::populate_grid(threadIdx.x + blockIdx.x * blockDim.x, config,
                          spacepoints, grid);
}
```

```
ALPAKA_FN_ACC void operator()(Acc const& acc,
    const seedfinder_config& config, sp_grid_view& grid_view,
    const spacepoint_collection_types::const_view& spacepoints_view) const {

    auto globalThreadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u];
    device::populate_grid(globalThreadIdx, config, spacepoints, grid);
}
```

Implementation of First Algorithm - Example

To show some examples, here is some comparisons between the CUDA and alpaka implementation for the this first spacepoint binning step.

Launching the setup kernel

```
kernels::populate_grid<<<num_blocks, num_threads>>>(
    m_config, spacepoints_view, grid_view);
CUDA_ERROR_CHECK(cudaGetLastError());
CUDA_ERROR_CHECK(cudaDeviceSynchronize());
```

```
alpaka::exec<Acc>(
        queue, workDiv,
        PopulateGridKernel{},
        m_config, spacepoints_view, grid_view
);
alpaka::wait(queue);
```

Implementation of First Algorithm - Results

This first pass basic test works! And wasn't that much work or overly complicated to setup. Here is a very early look at what it looks like when compared to two of the existing implementations. This table shows the average run time to run the spacepoint binning algorithm for one event, across a range of event types.

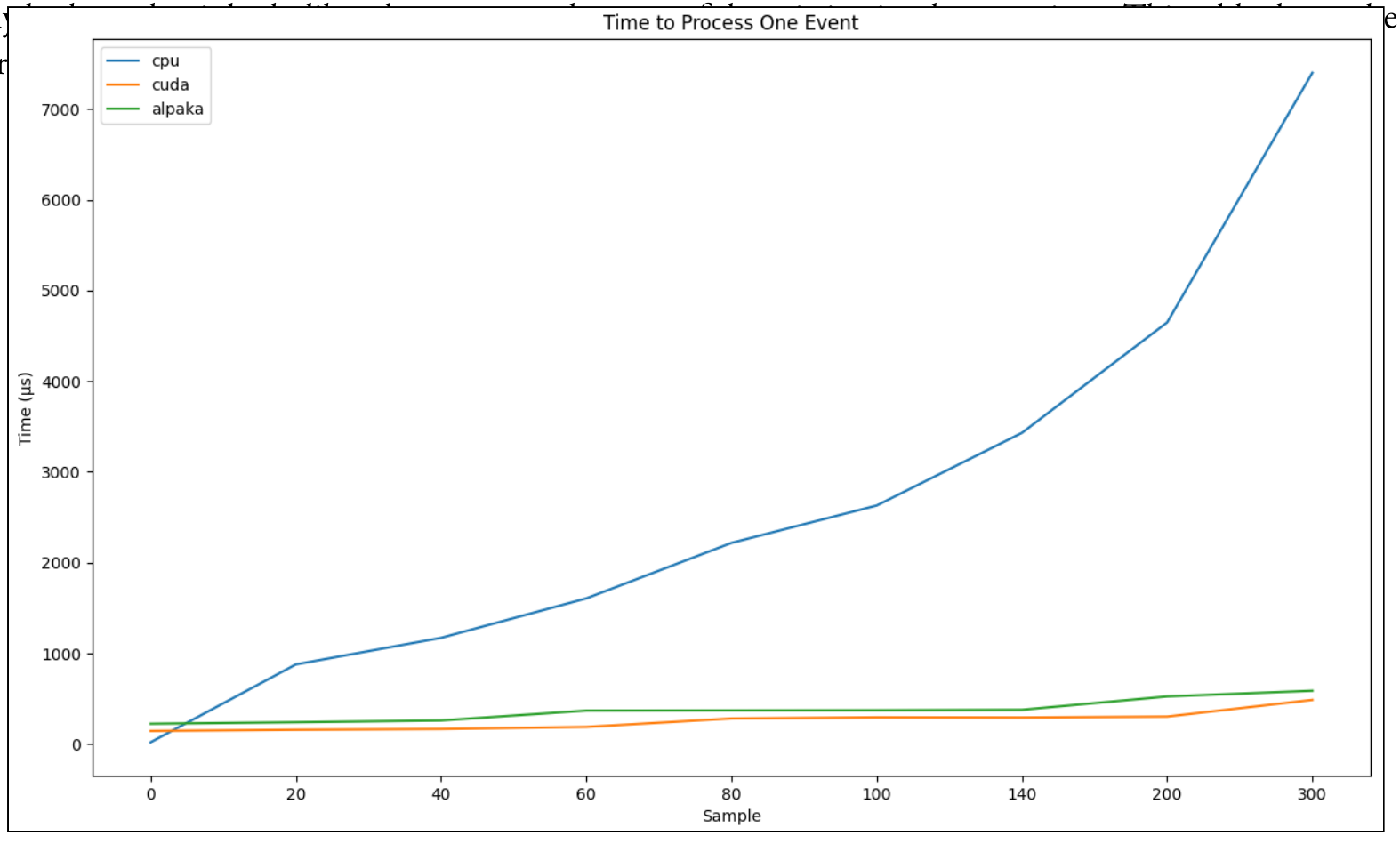
Time to Process One Event (μs)

Sample	CPU	CUDA	alpaka
single_muon	20	145	224
ttbar_mu20	878	158	241
ttbar_mu40	1171	166	260
ttbar_mu60	1606	189	369
ttbar_mu80	2218	282	371
ttbar_mu100	2630	295	373
ttbar_mu140	3431	293	378
ttbar_mu200	4649	303	525
ttbar_mu300	7400	487	588

Performed on a i9-10980XE and RTX A5000, average over 5 runs with 3 warm up runs.

Implementation of First Algorithm - Results

This first pass basic test works! And wasn't that much work or overly complicated to setup. Here is a very early [illegible] aver[illegible]e



Performed on a i9-10980XE and RTX A5000, average over 5 runs with 3 warm up runs.

Future Work

I've started work on getting the full seeding algorithm copied over to alpaka, which hopefully will give a more compelling demonstration of alpaka, and its performance characteristics, as well as the layout of the code in alpaka.

Along side that, there is a few remaining bugs and things to look at with this small test:

- I haven't tested across a wide-range of devices yet. I've done a quick CPU test, but not thoroughly checked yet. I think some parts may require slight re-organising to run seamlessly on both the CPU and the GPU[1].
- There is a race condition when running over multiple events: likely accessing part of the memory isn't fully setup/synchronised. Further debugging needed.
- There is still a performance delta between the CUDA and alpaka implementation, which is to be expected, but it would be interesting to look into it a bit more.

[1] alpaka tags alpaka kernels as `__device__` or `__host__` or both depending on the device. This currently doesn't play nice with part of I believe `vecmem`.

Conclusion

In Conclusion:

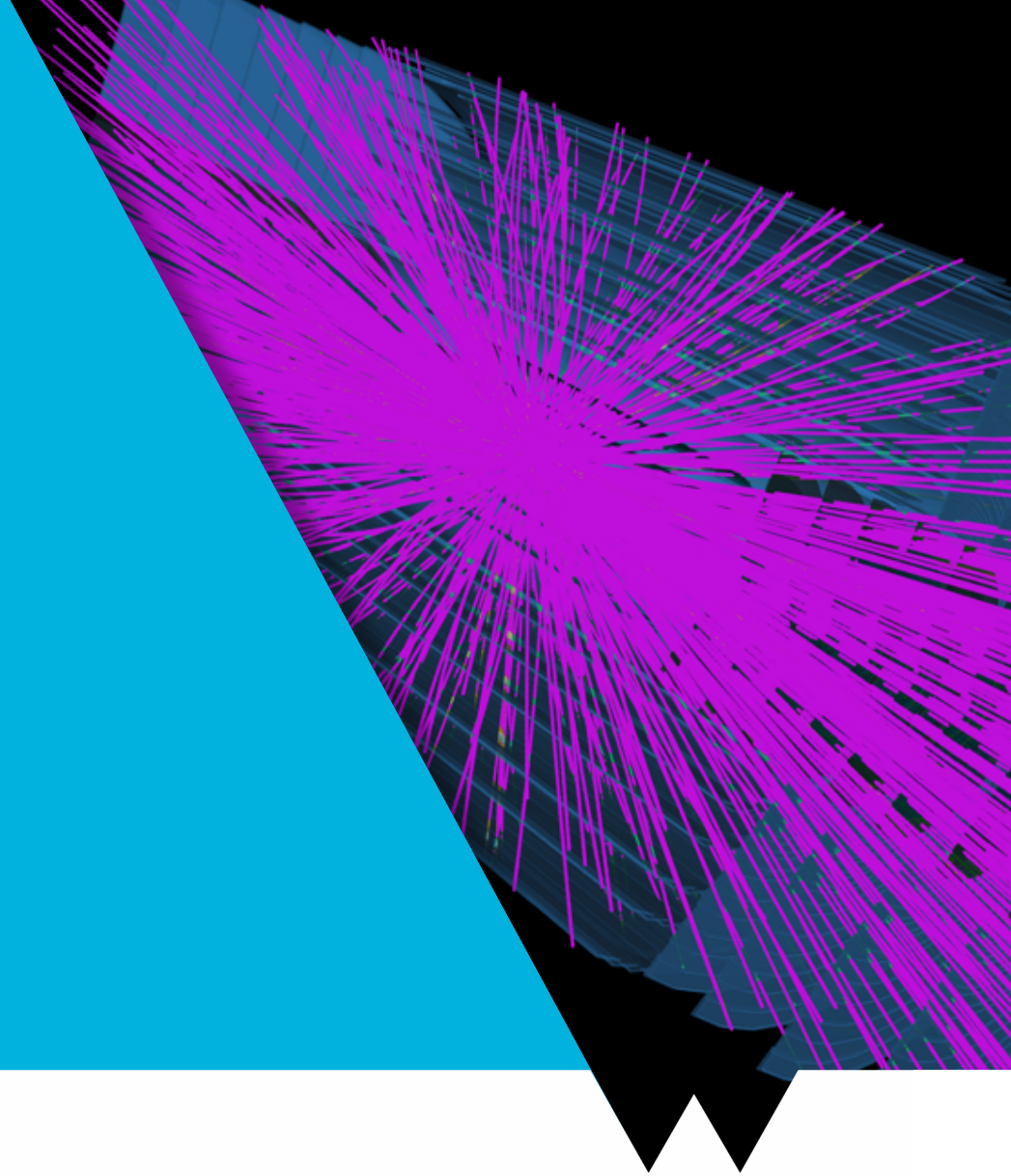
- traccc is a R&D effort as part of the ACTS project, working on exploiting GPUs and other accelerators to speed up tracking across a range of experiments.
- As part of that, many different acceleration abstraction libraries have been implemented, with alpaka being the newest.
- alpaka has good support already in HEP, and its parallelisation model make it a strong candidate for being the general purpose abstraction library.
- To that end, this talk outlines a first early test porting part of an algorithm to alpaka.
- Decent performance has been achieved, even without extensive tuning or testing of the different parallelisation parameters or threading model.
- Finally, there is a clear path forward to how to more extensively test alpaka, to understand how it performs and how easy it is to implement the sort of operations we need in it.

traccc & alpaka

Thoughts & First Look

Ryan Cross - GridPP49 and Swift-HEP

2023/03/30



WARWICK
THE UNIVERSITY OF WARWICK

Backup Slides

CUDA vs alpaka only